NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

LOSSLESS COMPRESSION USING BINARY NECKLACE CLASSES AND MULTIPLE HUFFMAN TREES
by

William L. Crowley Jr.

June 2001

Thesis Advisor:	Harold M. Fredricksen
Second Reader:	Craig W. Rasmussen

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** June 2001	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: Title (Mix case letters) Lossless Compression Using Necklace Classes and Multiple Huffman Trees			**5. FUNDING NUMBERS**
6. AUTHOR (S) Crowley, William L., Jr.			
7. PERFORMING ORGANIZATION NAME (S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME (S) AND ADDRESS(ES) N/A			**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public Release; Distribution is Unlimited			**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words)

In this thesis, we present two lossless compression approaches. Our Rotational Tree Approach (RTA) is based upon mathematics developed by Fredricksen. RTA uses the rotations associated with binary necklace classes to disperse source bit strings to a forest of Huffman encoding trees. Our Indexed Tree Approach (ITA) also uses a Huffman forest, but disperses bit strings via a simpler mechanism based upon the first few bits of each string. For text compression, we find RTA to be competitive with standard Huffman encoding while ITA is generally superior by a small margin of 1% – 3%. Both approaches owe their (limited) success to decreased modeling overhead as compared to standard Huffman encoding. Compression results against the Canterbury Corpus test suit and complete Java implementation code are included as appendices.

14. SUBJECT TERMS Lossless Data Compression, Discrete Mathematics, Analysis of Algorithms, Huffman coding, Rotational Tree, Index Tree.			**15. NUMBER OF PAGES** 174
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

LOSSLESS COMPRESSION USING BINARY NECKLACE CLASSES AND MULTIPLE HUFFMAN TREES

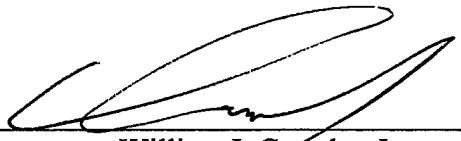
William L. Crowley Jr.
Captain, United States Army
B.S., Campbell University, 1991

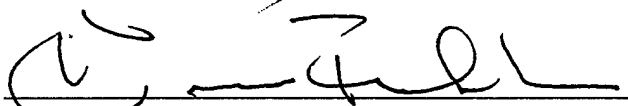
Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN APPLIED MATHEMATICS

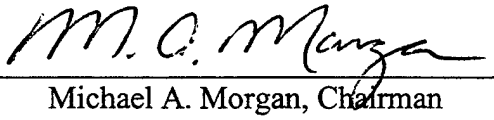
from the

NAVAL POSTGRADUATE SCHOOL
June 2001

Author: 
William L Crowley Jr.

Approved by: 
Harold M. Fredricksen, Thesis Advisor


Craig W. Rasmussen, Second Reader


Michael A. Morgan, Chairman
Department of Mathematics

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

In this thesis, we present two lossless compression approaches. Our Rotational Tree Approach (RTA) is based upon mathematics developed by Fredricksen. RTA uses the rotations associated with binary necklace classes to disperse source bit strings to a forest of Huffman encoding trees. Our Indexed Tree Approach (ITA) also uses a Huffman forest, but disperses bit strings via a simpler mechanism based upon the first few bits of each string. For text compression, we find RTA to be competitive with standard Huffman encoding while ITA is generally superior by a small margin of 1% – 3%. Both approaches owe their (limited) success to decreased modeling overhead as compared to standard Huffman encoding. Compression results against the Canterbury Corpus test suit and complete Java implementation code are included as appendices.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

ACKNOWLEDGMENTS

This thesis would not have been completed without the tireless efforts and tremendous production of Jason P. Marchant. He currently attends California Polytechnic State University, in San Luis Obispo, in pursuit of a Master of Science in Computer Science. Jason spent countless hours writing Java code and researching data compression techniques. Without his exhaustive efforts, we would not have been able to analyze the data and derive the conclusions that we present in order to complete our thesis.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A search for the string 'Huffman Encoding' in the academic science and engineering database *Ei Compendex Web* [1] yields the following results:

- 1970's database - 27 papers found
- 1980's database - 61 papers found
- 1990's database - 305 papers found

Clearly, despite the passing of nearly 50 years since D.A. Huffman's [2] landmark work on lossless compression, new uses are still being found for his ideas at ever increasing rates. The purpose of this paper is to present two previously untried approaches to lossless compression both of which involve the notion of multiple Huffman encoding trees.

Our Rotational Tree Approach (RTA) is based upon mathematics developed by Fredricksen [3, 4]. RTA uses the rotations associated with binary necklace classes to disperse source bit strings to a forest of Huffman encoding trees. Our Indexed Tree Approach (ITA) also uses a Huffman forest, but disperses bit strings via a simpler mechanism based upon the first few bits of each string. For text compression, we find RTA to be very competitive with standard Huffman encoding while ITA is generally superior by a margin of 1% – 3%.

THIS PAGE INTENTIONALLY LEFT BLANK

II. COMPRESSION FOUNDATIONS

A. COMPRESSING / DECOMPRESSING

In general, lossless data compression consists of reading a stream of symbols from file A, replacing each symbol with some (hopefully shorter) binary code, and writing the new codes to some file B. File B is now a compressed version of file A (or expanded, if the method was ineffective). The decompression process is the inverse. The codes stored in file B are read, converted to the original symbols, and written to some new file C. Upon successful completion of a lossless compression / decompression cycle, file C is identical to file A.

B. LOSSY VS. LOSSLESS

Data compression techniques can be divided into two broad categories – lossy and lossless. As the name implies, lossy compression techniques sacrifice some of the information content of the source file. Lossless compression techniques, on the other hand, preserve all the information content of the original file. Lossless techniques demand that the uncompressed file be bit for bit identical with the original source file.

Lossy compression is often used on files containing digitized voice or image data. Lossy compression makes sense in these settings since digital audio and video data files are always truncations of their original analog sources. Since one doesn't have all the data to begin with, it is often acceptable to sacrifice a bit more for the sake of compression (so long as the loss is not readily noticeable to the end user). Popular lossy compression schemes include JPEG (for images), MPEG (for video), and MP3 (for

audio). Lossy compression, while both useful and interesting, is not the subject of this thesis and will not be discussed further.

Lossless methods are used to compress documents, database records, executable files, and any other form of data that must be reproduced exactly. The two compression algorithms discussed in this paper are lossless.

C. INFORMATION THEORY

The theory of information developed by Claude Shannon [5] in the 1940's concerns the study of the storage, processing, and transmission of information. Information Theory provides an objective mathematical way of measuring the *information content* of a particular symbol (within a given message) and of the whole message. Information content is referred to as *entropy*, and is generally measured in bits. The entropy E of an input symbol i can be calculated using

$$E(i) = \log_2(1 / p(i)),$$

Equation 1

where p(i) is the probability of i occurring in the message. The entropy H of an entire message (typically a file) is then the sum of the entropies of all the individual symbols in the message, or

$$H = \Sigma(\log_2(1 / p(i))$$

Equation 2

summed over all the symbols in the message.

Consider the frequency distribution of characters in a simple 100-character message:

Characters Found	A	E	H	R	G	F
Frequency	45	16	13	12	9	5
Entropy	1.15	2.64	2.94	3.06	3.47	4.32

Table 1

The total entropy is H = (1.15 * 45) + (2.64 * 16) + (2.94 * 13) + (3.06 * 12) + (3.47 * 9) + (4.32 * 5) = 221.76 bits. Shannon's equations predict that under ideal circumstances we should be able to represent the above message in 222 bits. We contrast these statements with traditional ASCII encoding, which uses 8 bits per character or 8 * 100 = 800 bits to represent the data in the target file. One can immediately see how much room for improvement via compression actually exists in a typical text file. Thus, Shannon's methods provide us with a lower bound with which to measure the effectiveness of any compression scheme.

D. COMPRESSION STATISTICS

There are many ways to express the size reduction of a file after it has been compressed. In this paper we define the percentage of decrease (POD) via the formula

$$POD = (\textit{change in file size} / \textit{original file size}) * 100,$$

Equation 3

where the change in file size = original file size – compressed file size.

In the event that the compression fails (bloats the file) the resulting POD will be negative.

E. STATISTICAL VS. DICTIONARY MODELING

A compression model is a collection of data and rules used to map each input symbol to an output code. The term symbol is intentionally vague, as a symbol can represent an arbitrarily long string of bits deemed useful by the chosen model. A compression program uses its model to accurately define the probabilities for each input symbol. The model allows the program to produce an appropriate code for each symbol. Lossless compression is generally implemented using either statistical or dictionary modeling.

In statistical modeling the compression program analyzes the source file and finds the most common symbols. Typically, statistical modeling uses fixed-length symbols. Then each input symbol is assigned a replacement code. Replacement codes are normally short for the more common symbols and longer for the less common ones. An output file is then built by representing each symbol in the source file by its replacement code. When all the replacements are made the output file should be shorter than the source file since the more common symbols all get shorter representations. Even though some symbols get longer representations, they occur only infrequently in the text, thus they have a limited effect on the output file length.

In dictionary modeling a single code is used to replace a string of fixed-length symbols. Equivalently, one could say that symbols are of variable length. Regardless of the interpretation, a program using a dictionary model merely builds a list of the most frequent words, phrases, bit strings, or input symbols found in the source file. Each entry in the dictionary is then assigned a fixed-length replacement code, which can be thought of as simply an index into the dictionary.

The compression program uses its dictionary much like the authors of a thesis use the list of references at the end of the paper. Instead of writing out the full citation for a referenced paper, a number is used to indicate the citation's index within the bibliography. The reader of the thesis decompresses the citation by finding the indicated position within the list of references and mentally replaces the number with the full citation. Similarly, the dictionary compression program replaces input symbols with their assigned replacement codes (indices), and the decompression program does the inverse.

To draw a distinction between statistical and dictionary modeling, note that in statistical modeling fixed-length symbols are replaced by variable-length codes, but in dictionary modeling variable-length symbols are replaced by fixed-length codes.

F. STATIC VS. ADAPTIVE MODELS

The models in the previous section were described in a 'static' way, i.e., each model was created only after the *entire* source file was completely analyzed. This means two passes through the source file are required – the first to build the model and the second to do the actual symbol replacement. If a program is compressed using a static model then the decompression program must also have access to the same model in order to transform the codes in the compressed file back into the symbols from the original source file. In many cases, this means that the model will have to be included as part of the compressed file. Model overhead information is normally located at the beginning of the compressed file in an area referred to as the file header. Header overhead is one of the major performance burdens of static methods.

An alternative to static modeling is *adaptive* modeling (sometimes also referred to as dynamic modeling). In adaptive modeling the compression program updates the

model *as the output file is being created.* This means that the output codes produced by the compression program might have different meaning depending upon their relative position in the output file.

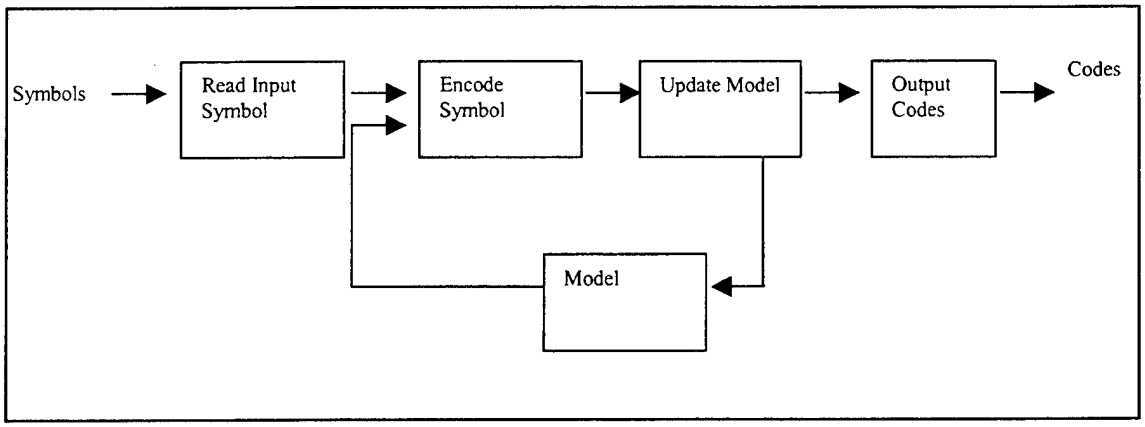


Figure 1: General Adaptive Compression [6]

There are several advantages to adaptive modeling. First, only one pass is required to encode the source file. Second, adaptive techniques quickly tune themselves to the file they are encoding and are thus better able to compress files containing markedly different types of data. Third, modeling information need not be explicitly included in the output file header as it is with static techniques. Instead, the adaptive decompression program simply begins with the same initial model as the compression program did, and then updates its model each time it reads a code from the compressed file. Thus, although the decompression program's model is in a constant state of change, it is always identical to the compression program's model at the same point in the process. Obviously, adaptive techniques rely on the compression and decompression

programs both beginning with the same model, and both updating the model in exactly the same fashion. The following diagram depicts the decompression process.

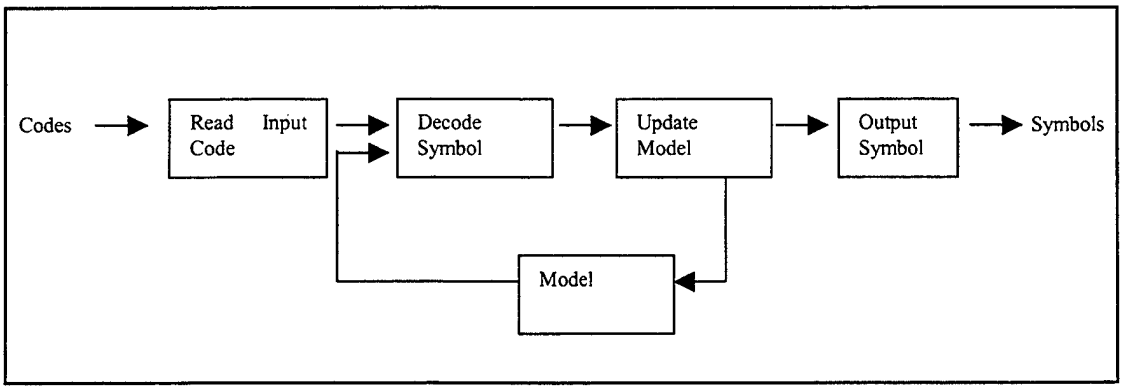


Figure 2: General Adaptive Decompression [6]

G. SURVEY OF LOSSLESS COMPRESSION TECHNIQUES

There are literally hundreds of serious data compression programs available on the web. The interested reader is referred to the excellent compression resource, "The Data Compression Library" [7]. We do not survey these compression implementations. Instead, we present the foundational techniques from which the current generation of compression programs has sprung. In this way we hope to paint a clear picture of the field of compression, and to show where our methods fit into the field.

A word of warning for those who do decide to explore the world of compression via the Internet: The vast majority of commercial applications are enhancements and hybrids of the more basic techniques, which follow in this section. These applications are typically tuned for specific data or specific performance (speed, memory footprint, or file size). Few limit themselves to any one compression technique. Instead, multiple

techniques are used in an effort to squeeze the maximum possible compression out of source files. Naturally, this leads to increased algorithm and program complexity. In short, the Internet may not be the best place to learn the fundamentals of compression. Two excellent books on the subject are "The Data Compression Book" [6] and "Compression Algorithms for Real Programmers" [8].

Most of the following techniques can use either a static or adaptive (dynamic) model. Interestingly, most statistical approaches seem to have been originally conceived in a static manner and later explored using adaptive methods. On the other hand, dictionary techniques typically began as adaptive methods, and only later began to be investigated statically. The techniques that follow are presented using the method with which they were originally developed.

1. Run Length

Long runs of repeated characters are one of the simplest forms of redundancy found in a file [9, 10]. Consider the string of repeated characters,

AAAAAAAABBBCCCCCCCDDDDEEEEEEEEE.

A run length encoding scheme might represent the string as

8A3B7C4D9E.

Unfortunately, typical text does not exhibit these types of patterns. There are some applications of this technique - the runs of black and white pixels on a fax image for example, but in general it is not particularly suited as a stand-alone method. Typical text compression using run length encoding is less then 5%. Of course, greater compression results can be achieved with specific data (e.g., fax images).

2. Shannon-Fano

This technique uses a static statistical model to build a binary tree in a top-down fashion. Each leaf of the tree represents an input symbol, while the path from the root node to a leaf gives the leaf's replacement code. The replacement codes generated by the algorithm share the following properties:

- Codes are of variable length.
- High probability codes are shorter then low probability codes.
- No code is a prefix of any other code from the same tree.

The Shannon-Fano algorithm [11] proceeds through the following steps:

1. Build a frequency distribution for the input symbols.
2. Sort the list of symbols in descending order by frequency.
3. Initialize the replacement code for each symbol to be null.
4. Divide the list in two parts with the total frequency count of the upper half being as close as possible to the total frequency count of the lower half.
5. Append a 0 to the replacement code of each symbol in the upper half and a 1 to the replacement code of each symbol in the lower half.
6. Recursively apply steps 4 and 5 to each partial list until each list contains exactly one symbol.

Although an excellent technique, Shannon-Fano encoding has generally been replaced by Huffman encoding, which is marginally superior.

3. Huffman

Standard Huffman encoding [2] uses a static statistical model to build a binary tree in a bottom-up fashion. The codes generated by the tree share all the properties of

those of those generated by Shannon-Fano and have the added advantage of being provably optimal. Optimal in this case means that there isn't a better set of *integral-length* replacement codes than those generated by the algorithm.

The Huffman algorithm proceeds as follows:

1. Build a frequency distribution for the input symbols.

2. Create a collection of single-node trees C each associated with a symbol and its frequency.

3. Let t_1 and t_2 be two trees with the lowest frequency in Collection C. Replace t_1 and t_2 with a single tree t_3, formed by attaching t_1 and t_2 as the left and right descendants of the new node. The frequency associated with t_3 is the sum of the frequencies of t_1 and t_2.

4. While more than one tree remains in C, repeat step 3.

Though optimal, Huffman encoding does not quite reach Shannon's predicted entropy limit. Consider the breakdown of our imaginary 100-character message and its associated Huffman tree shown below:

Symbol	Frequency	Entropy (bits)	Total Entropy (bits)	Huffman Code Length (bits)	Total Huffman Code Length (bits)
A	45	1.15	51.75	1	45
E	16	2.64	42.24	3	48
H	13	2.94	38.22	3	39
R	12	3.06	36.72	3	36
G	9	3.47	31.23	4	36
F	5	4.32	21.60	4	20
	100		221.76		224

Table 2: Frequency Table for 100-Character Message

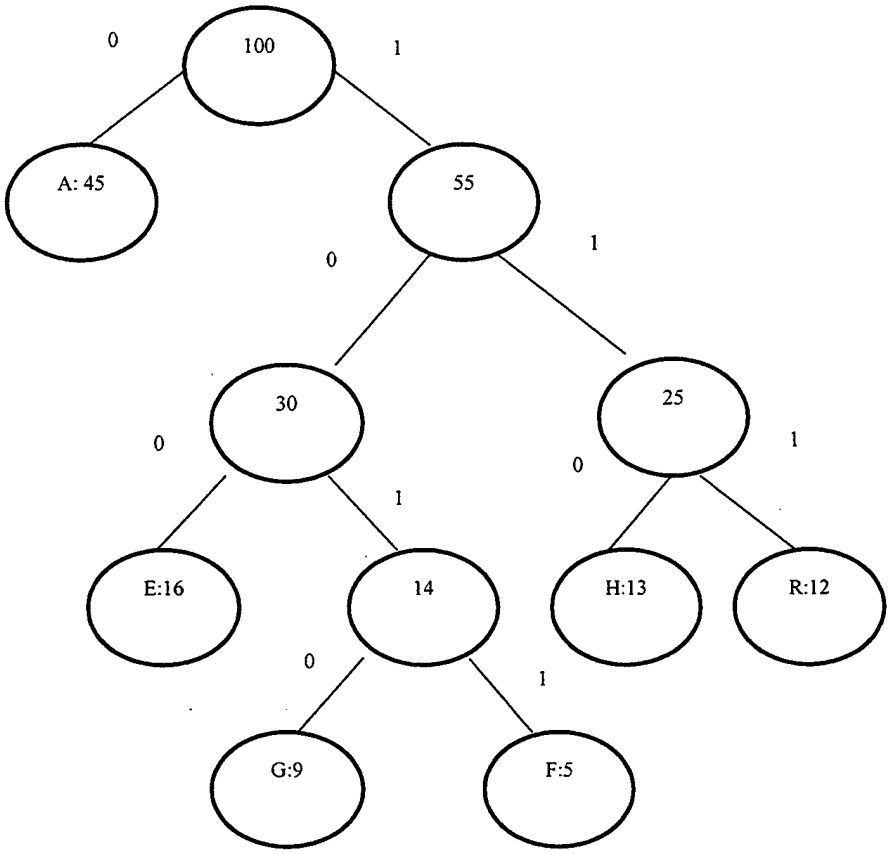


Figure 3: Huffman Tree

We see that while the total information content of the message is 221.76 bits, Huffman requires 224 bits to actually encode it. While this is not bad, it is not *truly* optimal. The deviation from the predicted value is due to the inability of the method to accurately represent the fractional entropy of each symbol. Thus, while there is no better integral length encoding scheme, there is still room to improve on Huffman's algorithm. Note that Shannon-Fano encoding will also require 224 bits for this file.

4. Arithmetic

Arithmetic coding [12, 13, 14] can use either a static or an adaptive statistical model. It completely discards Huffman's idea of replacing an input symbol with a specific code, and instead replaces the entire string of the input symbols in a message with a single floating point number between 0 and 1. For this reason it does not suffer the discrete code-length limitations of Huffman encoding. Arithmetic encoding uses a probability distribution to assign a proportionate range between 0 and 1 to each input symbol. It is typically able to approach the Shannon theoretical minimum - sometimes yielding improvements of up to 10% over standard Huffman encoding. The major drawback of the technique is that it is computationally intensive and therefore slow.

Arithmetic Encoding Algorithm:

1. Build a probability table for all the input symbols.
2. Establish proportionate lower bounds (L_i) and upper bounds (U_i) between 0 and 1 for each symbol based upon its probability of occurrence.
3. low = 0.0, high = 1.0, range = 1.0
4. Read in the next symbol to encode S_i.
5. range = high – low.
6. high = low + range * U_i.
7. low = low + range * L_i.
8. While more symbols remain to encode, repeat step 4.
9. Output the floating point number between low and high that has the shortest binary representation.

Consider a hypothetical message written in a 3-symbol alphabet D, A, X with the following probability distribution and bounds. Notice that each character is assigned the portion of the 0 to 1 range that corresponds to its probability of appearance.

Symbol Index (i)	Symbol (S_i)	Probability (P_i)	Lower Bound (L_i)	Upper Bound (U_i)
1	D	0.20	0	0.20
2	A	0.18	0.20	0.38
3	X	0.62	0.38	1

Table 3: Probability Table for Arithmetic Encoding

To see how the method works, we encode the message "DXXA" one symbol at a time. The first symbol "D" has a range from 0 to 0.20 so we know that the final encoded value for the message must fall within this range. Each time we add a new symbol we further restrict the range of our final value that represents the entire message. Thus, the next symbol "X" restricts us to the last 62% of the current range, or 0.124 to 0.2. The next symbol is another "X" so we again restrict the output to the last 62% of the current range, or 0.17112 to 0.2. The final symbol "A" limits us to between 20% and 38% of the current range, or 0.176896 to 0.1820944. Now we pick the value in the final range with the shortest binary representation. This is accomplished with some relatively straightforward binary arithmetic. In this case the shortest value turns out to be 0.1796875 = 0.0010111 base 2. The message is encoded as a unique floating-point value between 0 and 1. We require 7 bits to compress 4 symbols (about the same performance as Huffman on this particular example).

Consider the resulting partition of the number line shown below to see how any string in our alphabet could be similarly, yet uniquely, encoded. Notice that there is exactly one path to any given "leaf" of our fuzzy ternary tree.

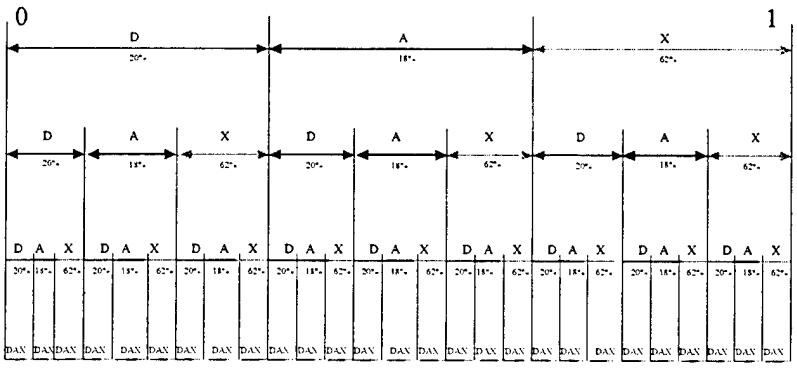


Table 4: Partitioning of the Number Line by Arithmetic Encoding: A Fuzzy n-ary Tree (intervals not to scale)

5. Ziv and Lempel

Jacob Ziv and Abraham Lempel are essentially the fathers of adaptive dictionary compression. Their first algorithm, commonly referred to as LZ77 [14], uses previously-seen text as a dictionary. The algorithm replaces variable-length phrases (symbols) from its input stream with fixed-length indices (codes) into its dictionary. What makes this an adaptive method is that its dictionary is literally a sliding window consisting (in a typical implementation) of the last 4k bytes of data from the input stream. Thus, while new groups of symbols are being read in to a *look-ahead buffer*, the algorithm is searching for matches between the look-ahead buffer and all strings located in the previous 4k bytes of data (the dictionary). If a match is found then an ordered triple (a, b, c) is sent to the output file. The triple consists of:

- An index a into the previous 4k of data.
- The length b of the match.
- The symbol c immediately following the match in the look-ahead buffer.

If no match is found then a and b are 0 and c is the first symbol in the look-ahead buffer. The text window and look-ahead buffer then slide forward sufficiently to shift the matching symbol(s) out of the look-ahead buffer and into the dictionary. The process then repeats.

The amount of compression achieved by this method depends upon the dictionary entries being sufficiently longer then the ordered triples they are replaced with. Here is a simplified example of the process:

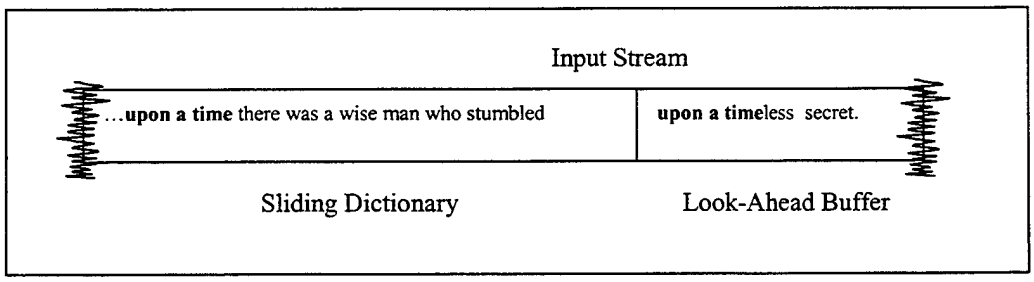


Figure 4: LZ77 Algorithm

Suppose our sliding dictionary is 64 bytes long and the look-ahead buffer is 16 bytes long. Further suppose that the first occurrence of the word "upon" begins at position 35 in the current sliding dictionary. Since a match has been found, our compression program will now send the ordered triple "35", "11", "l" to the output file. Since the dictionary is 64 bytes long 6 bits are required to represent each of the index and length of the match, plus 8 bits for the ASCII character "l". Thus, we have just encoded an 11 * 8 = 88 bit phrase in 6 + 6 + 8 = 20 bits.

There are two problems with LZ77. First, since the dictionary is constantly sliding, the algorithm sometimes actually throws away valuable old strings for worthless

new ones. This is a potential drawback to adaptive methods in general. Second, the length of pattern matches is limited to the length of the look-ahead buffer. Increasing the size of the look-ahead buffer, however, has a nasty side effect – it proportionately increases the number of string comparisons that must be performed against the dictionary. This creates a significant performance problem. Thus, LZ77 is not able to store long strings to its dictionary even when it would be highly profitable to do so.

A second adaptive dictionary technique developed by Ziv and Lempel, referred to as LZ78 [16], uses a different approach to building its dictionary. Instead of maintaining a sliding window into the preceding text, LZ78 incrementally builds its dictionary from strings found in *all of the preceding text*. The algorithm does this by *growing* dictionary entries one character at a time. For example, the first time the string "hello" is encountered it is not put in the dictionary. Instead, "h" is added to the dictionary. The second time "hello" is encountered it is still not put in the dictionary. Instead, "he" is added. The third occurrence puts "hel" into the dictionary, and so on until the fifth occurrence of the string "hello" actually causes "hello" to become part of the dictionary. This incremental procedure of slowly adding a string to the dictionary does a good job of eventually getting all of the frequently used strings into the dictionary. Once the dictionary begins to accurately model the source file, large savings can be achieved as the algorithm replaces long frequently occurring strings with short fixed-length indices.

LZ78 does not suffer from the same limitations as LZ77. It is able to handle long strings and once it adds a string to its dictionary it remains there until the entire file is processed. LZ78 has its own difficulties, however. In LZ77 the dictionary is easier to manage. It is a fixed block of already processed data. In LZ78 the dictionary starts with

nothing but the null string and grows. It must be managed with a data structure, (typically an n-ary tree that looks remarkably like the fuzzy tree shown earlier for arithmetic encoding) and limited to some finite size (typically around 2^{16} entries). Despite these problems LZ78, like LZ77, was a groundbreaking compression approach. Terry Welch later enhanced LZ78 – the resulting algorithm is commonly referred to as LZW.

THIS PAGE INTENTIONALLY LEFT BLANK

III. MATHEMATICAL FOUNDATIONS

A. INTRODUCTION

In this chapter we will discuss the necklace algorithm and the properties of the necklace algorithm that we find to be useful for compression. First we explore properties of binary n-tuple necklaces. Next, we consider equivalence classes of necklaces and how to enumerate the equivalence classes for a given n using the Burnside formula [17]. Then we use the Theta Algorithm [3] to generate all of the equivalent necklaces for a given n. Next, we consider the space reduction that can be realized from the sub-cyclic properties of necklaces. Finally, we describe a close relative to the necklace classes, the Lyndon Words.

B. NECKLACE CLASSES

The binary n-tuples are strings of 0's and 1's of a fixed length n. The circular rotation of a binary n-tuple produces a cyclic equivalence class of binary n-tuples called a necklace class. We represent each necklace class by the numerically largest n-tuple in the class. This gives us Z[n] necklaces representing the 2^n strings from 000...0 to 111...1, where Z[n] is defined in equation 4. The necklace algorithm provides an efficient way to list these necklace classes. The necklace algorithm typically produces the necklaces in the order from largest to smallest. There is no difference in which way we represent these strings and an equivalent version of the necklace algorithm could invert the order of the necklace classes.

1. Equivalence Classes and the Burnside Formula

A necklace is an equivalence class of strings of bits of length n that can be rotated to obtain equivalent aspects of the same necklace. Thus, two necklaces are said to be inequivalent if, no matter how rotated, one cannot be transformed into the other. Z[n] gives the number of inequivalent necklaces of length n, where Z[n] is determined by using an instance of the Burnside enumeration formula:

$$Z[n] = (1/n)\Sigma(\phi(d)*2^{(n/d)}), \text{ for } d = 1 \text{ to } n \text{ and } d \text{ is a divisor of } n.$$

Equation 4

Here $\phi(d)$ is Euler's Totient Function, defined as the number of positive integers less than d and relatively prime to d.

The formula in equation 4 enumerates the necklace classes for a given n, and the necklace algorithm tells us what the necklaces are. To illustrate the use of the formula we consider an example:

Let n = 6, then

$$Z[n] = 1/6(1*2^6 + 1*2^3 + 2*2^2 + 2*2) = 1/6\ (64+8+8+4) = 84/6 = 14.$$

d=1 d=2 d=3 d=6

When d = 1, only 1 is relatively prime to 1. When d = 2, again only 1 is relatively prime to 2. When d = 3, 1 and 2 are relatively prime to 3. When d = 6, both 1 and 5 are relatively prime to 6.

We identify two upper bounds for the number of necklace classes of length n. A simple upper bound is $(2^{(n+1)})/n$. A tighter upper bound is $2^{((n+1)-\lceil \log_2 n \rceil)}$. When n = 6, we

know that there are 14 necklace classes. If we compute the first upper bound we get 21.33 while the second upper bound yields 16. We use the tighter upper bound in the implementation of our compression algorithms.

2. Necklace Algorithm

The necklace algorithm is fairly straightforward. First we need to define the θ step. This operation θ: $V^n \rightarrow V^n$ is defined as follows: $\theta(a_1,a_2, \ldots, a_n) =$ $(b_1, b_2, \ldots, b_n)$ where $b_i = a_i$ for i =1, 2, ..., j-1 and j is defined as the largest subscript such that $a_j > 0$ and $a_k = 0$ for all $k > j$. Then

$$b_j = a_{j-1}, b_{j+t} = b_t \text{ for } t = 1, 2, \ldots, n - j.$$

Necklace Algorithm (for necklaces of length n) [3]

0. The initial necklace is $11\ldots1 = 1^n$;

1. To find the $i+1^{st}$ necklace apply θ to the i^{th} necklace;

2. The resulting string is the next necklace if and only if j|n.

3. If j does not divide n, then apply $\theta^2, \theta^3, \ldots, \theta^k$ to the i^{th} necklace until the smallest k is found so that j|n. The resulting string is the next necklace.

4. If the necklace found is not the last necklace, namely, 0^n, return to step 1.

Note: If step 2 is modified such that j = n, we produce a list of all Lyndon words of length n (Lyndon words are discussed in more detail in paragraph B).

To illustrate and to clarify the algorithm, start with the largest n-long necklace (an n-long string of 1's), subtract one from the last 1 in the string and copy the string $a_1 \ldots a_{j-1}$ until an n-long string is formed. Now do the "j divides n check". If j|n then the string

formed is a necklace, if not, subtract one again from the last non-zero in the current string and copy to form an n-long string.

As an example consider the necklaces produced when n = 6:

111111	This is the first necklace ($a_1, a_2, \ldots, a_6$).	
111110	j = 6	Since 6 divides 6, this is a necklace.
111101	j = 5	Since 5 does not divide 6, this is not a necklace.
111100	j = 6	Since 6 divides 6, this is a necklace.
111011	j = 4	Since 4 does not divide 6, this is not a necklace.
111010	j = 6	Since 6 divides 6, this is a necklace.
111001	j = 5	Since 5 does not divide 6, this is not a necklace.
111000	j = 6	Since 6 divides 6, this is a necklace.
110110	j = 3	Since 3 divides 6, this is a necklace.
110101	j = 5	Since 5 does not divide 6, this is not a necklace.
110100	j = 6	Since 6 divides 6, this is a necklace.
110011	j = 4	Since 4 does not divide 6, this is not a necklace.
110010	j = 6	Since 6 divides 6, this is a necklace.
110001	j = 5	Since 5 does not divide 6, this is not a necklace.
110000	j = 6	Since 6 divides 6, this is a necklace.
101010	j = 2	Since 2 divides 6, this is a necklace.
101001	j = 5	Since 5 does not divide 6, this is not a necklace.
101000	j = 6	Since 6 divides 6, this is a necklace.
100100	j = 3	Since 3 divides 6, this is a necklace.
100010	j = 4	Since 4 does not divide 6, this is not a necklace.
100001	j = 5	Since 5 does not divide 6, this is not a necklace.
100000	j = 6	Since 6 divides 6, this is a necklace.
000000	j = 1	Since 1 divides 6, this is a necklace.

There are fourteen necklaces generated by the Necklace Algorithm; that is the number given by the equation 4. Additionally, it is easy to see that a string that ends with a 1 (except all 1's, of course) can never be a necklace, since that 1 could be rotated to the

front of the n-tuple and the rotated n-tuple, being larger, will have already appeared on a previous necklace in the lexicographic listing of the necklaces. For each value of j, the string $a_1, a_2, \ldots, a_{j-1}$ is a Lyndon word of length j. (See paragraph C for a discussion of Lyndon words.)

3. Necklace Algorithm Pseudo Code

DESCRIPTION: Finds all the necklace classes of length n. Note the indexing scheme used by the bit arrays of this method. The most significant bit is considered to be at index 1 while the least significant bit is at index n.

PRECONDITION: The array classes are large enough to hold all the necklace classes generated by the algorithm.

POSTCONDITION: classes contains all the necklace classes of length n sorted in descending numerical order.

```
void getNecklaceClasses(/* IN  */ int n,
                        /* OUT */ BitArray[] classes)
{
   BitArray c = 2^n - 1;      // the first class is all 1's
   classes[0] = c;            // store it at index 0 of classes
   int k = 1;                 // index to store next class at
   while (c > 0)
   {
      int j = c.leastSig1();  // j gets index of least sig 1 in c
      c[j++] = 0              // replace least sig 1 in c with 0
      int i = 1;
      while (j <= n)          // copy over, copy over, ...
         c[j++] = c[i++];
      if (j | n)              // j-check, if TRUE it's a class
         classes[k++] = c;
   }
}
```

Figure 5: The Necklace Algorithm Pseudo Code

4. Sub-cyclic Properties

As mentioned previously, the necklace algorithm generates all of the necklace classes for a given n. These necklaces are listed lexicographically from the largest (1^n) to the smallest (0^n). This is contrary to normal order, but the results are equivalent to an ordering from smallest to largest and it is easy to translate between the necklaces listed from smallest to largest.

One of the most interesting properties of the necklace algorithm is that the sub-cyclic rotations also occur for each divisor of a given n. That is, all the m-long necklaces appear where m is a factor of n. The factors of n tell us the length of the sub-cyclic or repeating strings for the given n. Using the Mobius function [17], in place of the totient function, one may enumerate the sub-cyclic rotations for the given factors of n. In fact, regardless of the value of n, the number of sub-cyclic rotations of length m will always be the same for the factors m of n. In other words, since 1 is a factor of every n, there are two sub-cyclic rotations of length 1, namely 0 and 1 and the necklace (10) appears in every necklace class when n is even. Thus, when m|n, each m-long necklace will appear in the necklace algorithm for the parameter n.

In the example given where n = 6 and has the factors 1, 2, 3 and 6, the necklaces that appear are all the necklaces for m = 1, 2, 3 and 6. If we modify the enumeration formula in equation 4 by substituting for $\theta(d)$ the function $\mu(d)$, where $\mu(d)$ (the Mobius function) is defined by

$\mu(1) = 1$;

$\mu(n) = (-1)^k$ when n has k distinct prime factors;

$\mu(n) = 0$ when n has a square or higher order factor, we obtain the expression on the right hand side of equation 5.

We obtain the number of necklaces of length n by applying this modified enumeration formula on each of these factors m of n.

$$N(m) = (1/m)\Sigma(\mu(d)*2^{(m/d)}).$$

Equation 5

So, when n is 1, there are 2 necklaces; when n is 2, there is 1 necklace. We determine the number of necklaces when n is 3 to be $1/3(1*2^3 - 1*2) = 1/3(8-2) = 6/3 = 2$. These are the necklaces (110 and 100). Finally, we determine the number of necklaces when n is 6 as $1/6(1*2^6 - 1*2^3 - 1*2^2 + 1*2) = 1/6(64-8-4+2) = 54/6 = 9$. Therefore, 9 of the 14 necklace classes will be of length 6 as can be determined from our listing above of the necklace algorithm. Thus the total number of necklaces, including all sub-cyclic necklaces, for n = 6 is (2+1+2+9) = 14.

These sub-cyclic rotations are important in our version of the data compression algorithms, because, it is cheaper to represent a string with fewer rotations, than one with greater rotations. Namely, it costs only 1 bit to represent a necklace with sub-cyclic length 2, where it costs 3 bits to represent all of the possible rotations of a necklace of length 6.

C. LYNDON WORDS [18]

Although the two lossless compression techniques discussed in this paper do not use Lyndon words, we cover them because of their relation to necklaces and because we recommend them for future research.

A *k*-ary *necklace* is an equivalence class of *k*-ary strings under rotation. We take the lexicographically smallest such string as the representative of each equivalence class and use this in the output of the program. A *Lyndon word* is an aperiodic necklace representative.

The illustration below shows the 6 binary necklaces with 4 beads and the corresponding equivalence classes of strings. The three Lyndon words are 0001, 0011, and 0111.

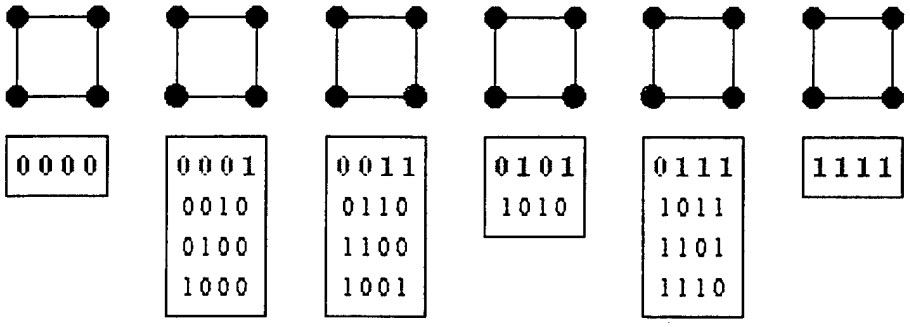


Figure 6: Lyndon words of length four.

Equation 6 gives the explicit formula for the number of Lyndon words of length n over a *k*-ary alphabet.

$$N_k(n) = (1/n)\Sigma(\mu(d/n)*k^d)$$

Equation 6

IV. ROTATIONAL TREE APPROACH

A. INTRODUCTION

In this chapter we exhibit the bulk of our research efforts. Our efforts to find a compression application involving the binary necklace classes discussed in section III, led us to The Rotational Tree Approach (RTA). Our motivation for exploring the approach arises from the following observation. Let A be the set of all bit strings of length n. Further, define B to be the set of all necklace classes of length n. Consider the onto mapping from the domain A to the codomain B, where each n-tuple maps to its representative necklace class in B:

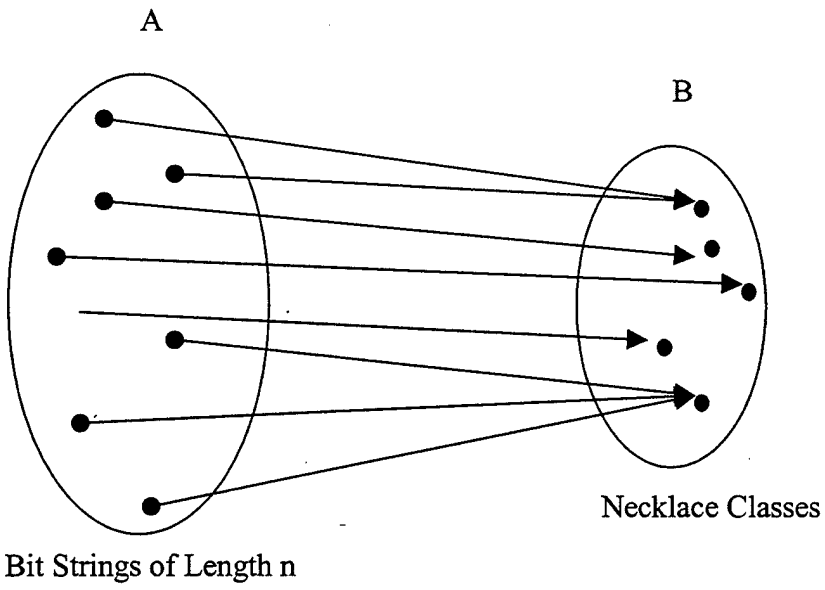


Figure 7: Mapping of Bit Strings to Their Necklace Class

This mapping is attractive from a compression standpoint because there is a significant space reduction between the number of bit strings and the number of necklace

classes to which they map since there are 2^n bit strings of length n and less than $2^{(n+1)} / n$ necklace classes.

n	Bit Strings	Classes	Decrease
6	$2^6 = 64$	14	78%
12	$2^{12} = 4,096$	352	91%
16	$2^{16} = 65,536$	4,116	94%
20	$2^{20} = 1,048,576$	52,488	95%
24	$2^{24} = 16,777,216$	699,252	96%
28	$2^{28} = 268,435,456$	9,587,580	96%
32	$2^{32} = 4,294,967,296$	134,219,796	97%

Table 5: Space Reduction Afforded by Mapping Strings to Necklace Classes

B. EXPLANATION

In order to take advantage of the resulting space reduction, our idea is to map each source symbol in A to its corresponding necklace class in B, and then represent that class with an index. We define the index i of an n-bit class as the position of the class in the ordered list of all n-bit classes as given by the Necklace Algorithm. Thus, by mapping a symbol to an index class we can effectively decrease the number of distinct symbols as well as the size of each such symbol. This sounds promising.

The difficulty is that there are many n-bit strings that map to the same index class. Thus, in order to reverse the mapping, which is necessary for decompression, additional information is needed; specifically, we need to know the number of rotations r originally used to transform the bit string to its necklace class.

What we have at this point is a vague model involving the quantities r (rotation) and i (index). What we need is a clever way to use them. As r and i are fixed in length (with respect to a given n) static Huffman encoding seems a good choice.

Our initial efforts revolved around a two-Huffman-tree approach. We proceed as follows: We first read n-bit symbols from a source file. We then break each symbol into an index i and a rotation r. We build one Huffman tree using all the r's and another using all the i's. Finally, we write the Huffman codes for each r, i pair to the output file, in the order in which they were first discovered.

Surprisingly (for us at least) this approach is a complete failure. It typically achieves very low levels of compression (often even bloating the source file). After some analysis involving comparisons against a standard single Huffman tree built with n-bit symbols, we draw the following conclusions concerning this failed two-tree approach:

- The i-Huffman tree is much narrower and shallower than a standard Huffman tree built with n-bit symbols. This means the codes produced by the i-Huffman tree are shorter. This is exactly the type of decrease in symbol number and size that we are looking for.
- The amount of statistical data (output file header) required for the two-tree approach is significantly less than that required for standard Huffman (more on this later).
- The added expense of the r-codes paired with every i-code completely erases the gains cited above.

Essentially, the r-codes are an added expense that the model cannot afford. What we need is a way to make the r-codes either smaller or else somehow more useful to the model. Our solution consisted of developing an approach involving multiple Huffman trees. It proceeds as follows: We first read n-bit symbols from a source file. Then we break each symbol into a rotation r and an i. We put the r's into one Huffman tree, and distribute the i's to a forest of Huffman trees based upon their associated r's. Finally, we write the Huffman codes for each r, i pair to the output file, in the same order in which they were discovered.

This approach puts the r-codes to work by using them to distribute the i-codes to not one, but many Huffman trees. This is advantageous, as each tree in the forest will be both narrower and shallower than the original i-tree, and as such will produce shorter codes. The next section refines this procedure.

C. ALGORITHM

1. Build a static statistical model of the data as follows:
    a. Construct n + 1 empty frequency tables (indexed from 0 to n) where n represents the length in bits of the symbols read from the input stream.
    b. Read the next n-bit symbol S from the input stream.
    c. Determine the number r of rotations required to transform S into its necklace class representative c.
    d. Determine the index i of c in an ordered list of all n-bit necklace classes.
    e. If i does not exist in frequency table r, put i into frequency table r with a frequency count of 1. Else, increment the frequency count of i in frequency table r.

f. If r does not exist in frequency table n, put r into frequency table n with a frequency count of 1. Else, increment the frequency count of r in frequency table n.

g. While there are more symbols in the input stream, repeat step *a*.

2. Create a Huffman tree from each of the n + 1 frequency tables. The leaves of Huffman tree n represent rotations. The leaves of all other Huffman trees represent class indices.
3. Reset the input pointer to the beginning of the input stream.
4. Read the next n-bit symbol S from the input stream.
5. Determine r and i for S.
6. Write the Huffman code for r in Huffman tree n to the output stream.
7. Write the Huffman code for i in Huffman tree r to the output stream.
8. While there are more symbols in the input stream, repeat step 4.

D. EXAMPLE

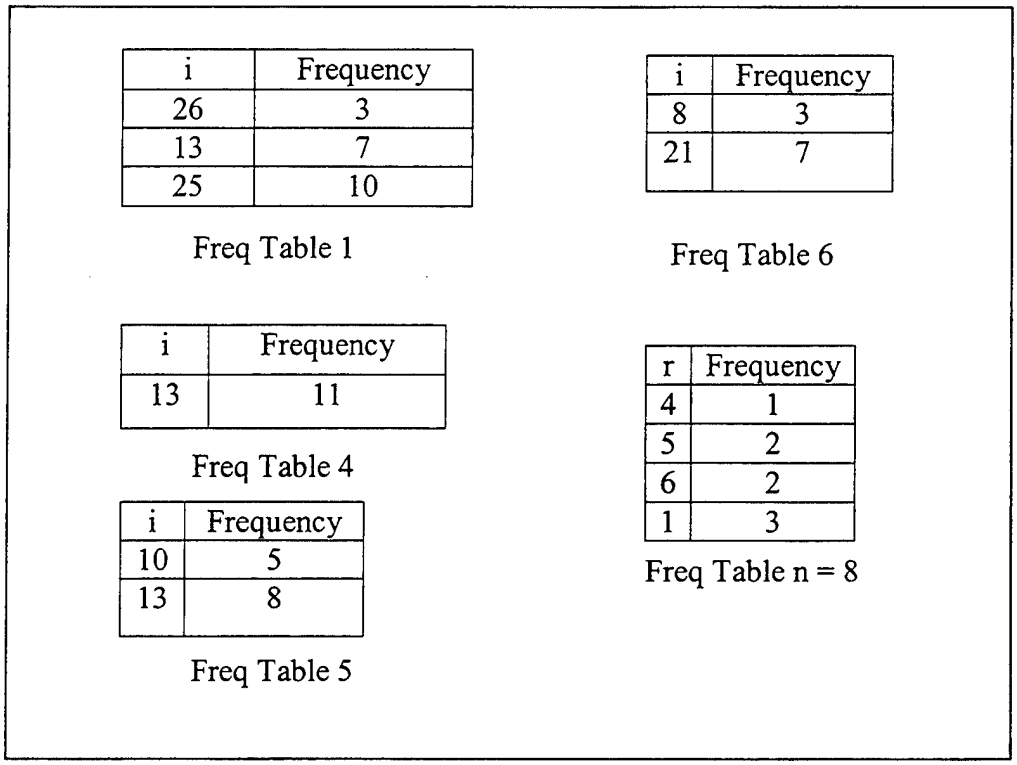
Consider a hypothetical 54-character message and its resultant breakdown:

Message: acgCs7bnacgCs7bnacgCs7bnggCCCCssss77777bnbnbnbnbnbnbnn

Symbol	Binary Representation	Necklace Class	r	i	Frequency
a	01100001	11000010	1	26	3
c	01100011	11011000	6	18	3
g	01100111	11101100	5	10	5
C	01000011	11010000	6	21	7
s	01110011	11100110	1	13	7
7	00110111	11100110	5	13	8
b	01100010	11000100	1	25	10
n	01101110	11100110	4	13	11
					54

Table 6: Example Message Breakdown by RTA (n = 8)

Step 1 of the RTA algorithm constructs the following frequency tables:



i	Frequency
26	3
13	7
25	10

Freq Table 1

i	Frequency
8	3
21	7

Freq Table 6

i	Frequency
13	11

Freq Table 4

r	Frequency
4	1
5	2
6	2
1	3

Freq Table n = 8

i	Frequency
10	5
13	8

Freq Table 5

Figure 8: RTA Step 1: Build Frequency Tables

Step 2 of the RTA algorithm constructs a Huffman tree for each frequency table.

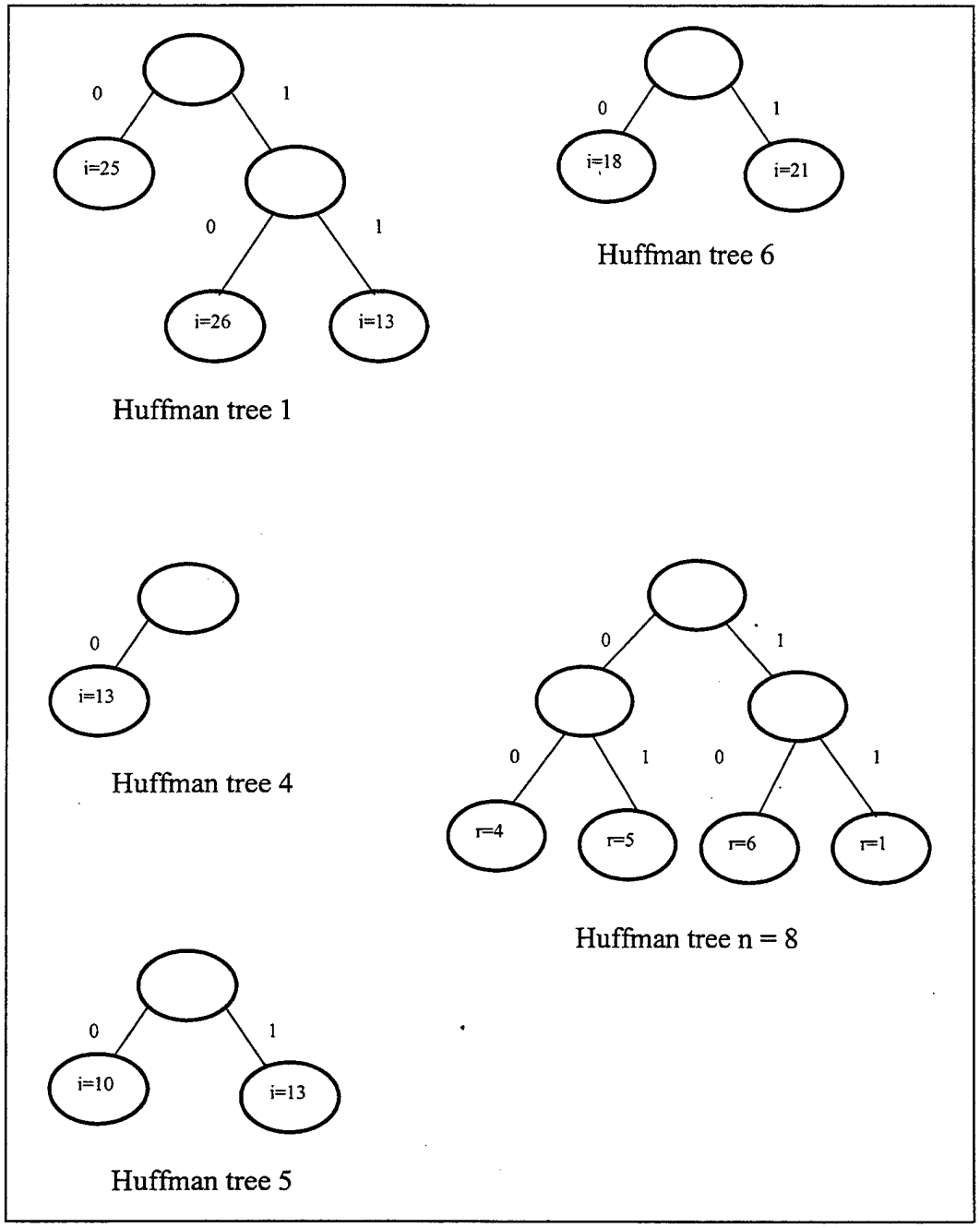


Figure 9: RTA Step 2: Build Huffman Trees

With the Huffman trees constructed, steps 3 thru 8 of the RTA algorithm begin the process of creating the output codes. This is accomplished by generating the r, i pairs to replace each input symbol in the message. Thus, the stream of input symbols

acgCs7bna...

is transformed into the stream of r, i pairs:

1, 26 6, 18 5, 10 6, 21 1, 13 5, 13 1, 25 4, 13 1, 26...

which then becomes the output stream:

11, 10 10, 0 01, 0 10, 1 11, 11 01, 1 11, 0 00, 0 11, 10...

by replacing each r with its Huffman code from tree n = 8 and each i with its Huffman code from tree r. For example, the r, i pair that corresponds to symbol "a", (found on table 6 above) is 1, 26. The code that represents r from the n=8 tree (figure 9 above) is 11. The code that represents the i component from Huffman tree 1 (figure 8 above) is 10. Therefore the input symbol "a" becomes the output 1110. Now move to the next symbol in the stream and repeat the process.

F. INTERPRETATION OF RESULTS

We successfully implemented RTA in the Java programming language (see Appendix A) and collected empirical data (see Appendix B). The overall compression performance of RTA is roughly equivalent to that of standard Huffman encoding on both text and uncompressed bitmapped image data. Computationally, RTA is more expensive then Huffman encoding, but not prohibitively so. However, our implementation of RTA is space-hungry – requiring up to $5*2^n + (4*2^{(n+1)}) / n$ bytes just to build the tables needed to efficiently calculate r and i for all the input symbols. This calculation does not

include the memory needed to build the actual Huffman trees, which is data dependent but can also be very significant. The compression results for RTA are in fact cut off at n = 24 due to memory constraints - a typical 0.5 MB file requiring 86 MB of table building memory alone!

The compression performance of RTA is most easily explained by once again considering the failed two-tree approach discussed in part B above. By comparison with standard Huffman encoding, the two-tree approach has a favorable header size, and a favorable i-tree width and depth. Its fatal flaw is the overhead of the r-codes paired with every i-code. RTA is strapped by the same r-code overhead, but unlike the two-tree method, RTA uses its r-codes advantageously. By building a forest of i-trees instead of a single i-tree, RTA is able to generate much shorter i-codes. This allows the RTA forest to *approach* the performance of a single optimal Huffman tree - even with the added overhead of the r-codes. A useful way to view the Huffman forest created by RTA is as a *single distributed pseudo-Huffman tree* with the r-tree as the root, and the i-trees as leaves.

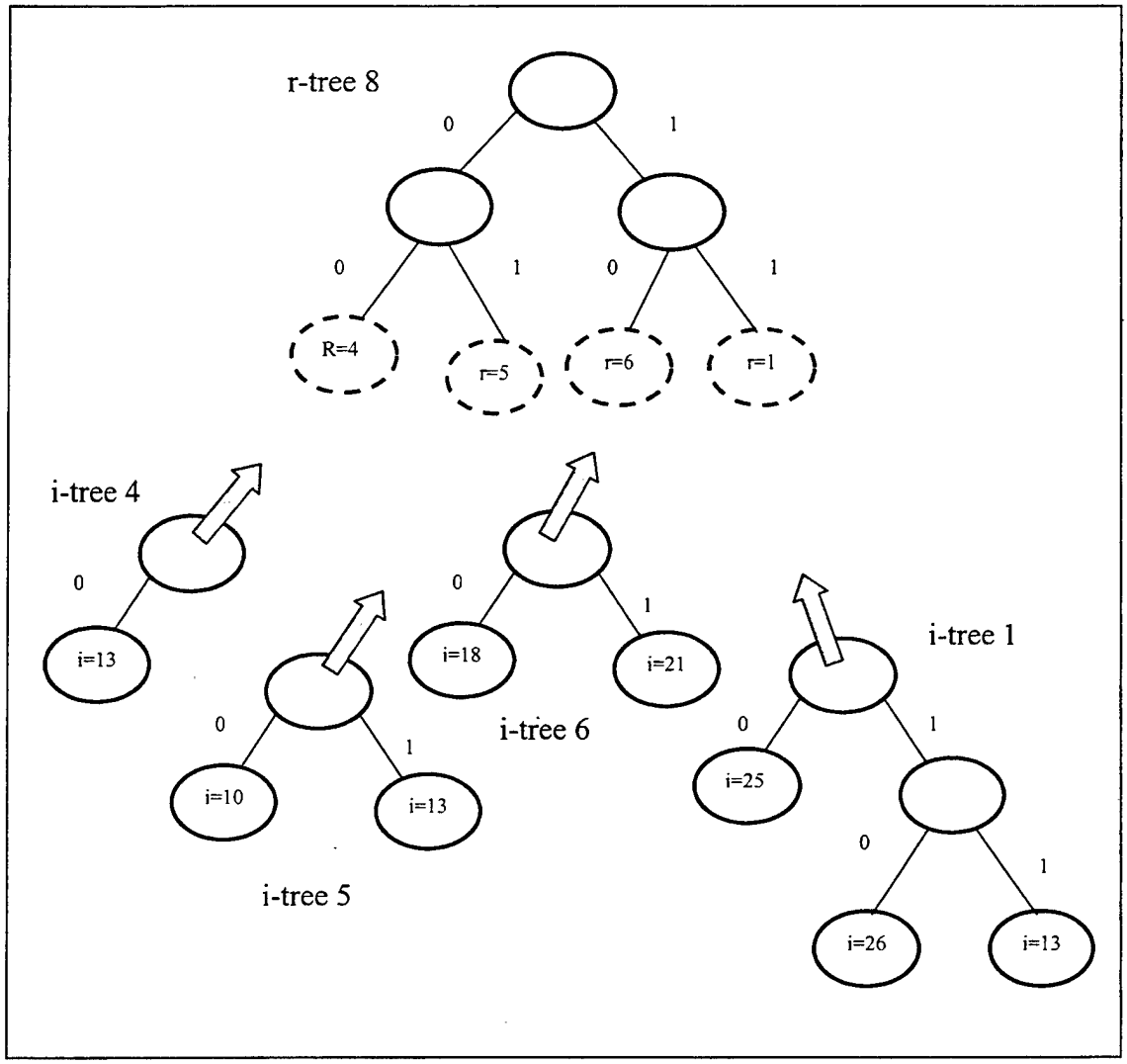


Figure 10: Distributed Pseudo-Huffman Tree Created by RTA

It is easy to see that this particular distributed tree is not a true Huffman tree and is therefore not optimal. Note that node i=10 from i-tree 5 is higher in the distributed tree than node i=13 from i-tree 1, even though node i=13 has a greater frequency.

After studying the tree breakdowns of our empirical data (see Appendix D for an example) we conclude that the weight of the distributed tree created by RTA is typically within 5% of the weight of an optimal Huffman tree. While it is theoretically possible for

the distributed tree to be a true Huffman tree, it would be an extremely rare occurrence when compressing real data.

Using this insight into the distributed tree of RTA, we conclude that the sum of the lengths of the i-codes and r-codes produced by RTA will approach the sum of the codes produced by a standard Huffman tree built from the same data. The problem is, if RTA can only approach Huffman encoding, then how is it that the empirical data shows RTA marginally outperforming Huffman encoding on almost half of the files in the test suite? The answer lies in the compression header. RTA requires less overhead than standard Huffman encoding to pass the statistical model from the compression program to the decompression program.

Model overhead is one of the drawbacks of static methods like Huffman encoding, the two-tree approach, and RTA. However, without a statistical model the decompression program has no way of uncompressing the codes in the compressed files back into the symbols of the source file, so inclusion of the compression model is necessary. Our implementation of Huffman, two-tree, and RTA all pass their static models to the decompression program in the same way. Each includes a frequency ordered list of all the leaves of their Huffman tree(s) in their respective header. The header includes other information as well (see Appendix E for details), but none so voluminous as the leaf list.

The symbols we shall encode in our Huffman code are binary n-tuples. A first pass through the data determines their probability of occurrence in the text. Armed with these probabilities a Huffman code can be made for the file under consideration and the leaves in the Huffman tree generated are the n-tuples appearing in the file. Then, in the

worst case, the leaf list for Huffman encoding will contain 2^n entries of n bits each. This means the leaf list for a standard Huffman tree can require up to $n * (2^n) / 8$ bytes of space.

In order to gain insight into the amount of space consumed by the leaf lists of RTA, we again turn to our failed two-tree approach. We noted earlier that two-tree had a significantly smaller header than standard Huffman encoding. This is a consequence of the space reduction afforded by using necklace class indices as leaves of the i-tree. Even though two-tree must put two leaf lists in its header, one for the r-tree and another for the i-tree, we see that the two-tree method still requires much less overhead than standard Huffman. The length and number of the leaves in the leaf lists of the two-tree method are governed by the properties of binary necklace classes discussed in chapter 3. The relevant information is summarized in the table 17.

	Leaf Length (bits)	Upper Bound on Number of Leaves
r-tree	$\lceil \log_2 n \rceil$	n
i-tree	$n + 1 - \lceil \log_2 n \rceil$	$2^{(n + 1 - \lceil \log_2 n \rceil)}$

Table 7: Leaf List Properties

An upper bound U on the maximum leaf list size (in bytes) for the two-tree approach is found by summing the product of the leaf length and the number of leaves for each leaf list then dividing by 8, or

$$U = (\lceil \log_2 n \rceil * n + (n + 1 - \lceil \log_2 n \rceil) * 2^{(n + 1 - \lceil \log 2 n \rceil)}) / 8.$$

The following table points out the drastic differences *possible* between the leaf lists of standard Huffman and those of the two-tree approach.

n	4	8	12	16	20	24
Standard Huffman	8	256	4,096	65,536	1,048,576	16,777,216
Two-Tree Approach	4	51	582	13,320	131,085	2,621,455

Table 8: Worst Case Leaf List Comparisons

When run on typical data, the actual differences between the leaf lists of the two approaches is never as dramatic as the worst-case scenario shown above. If it had been, two-tree would likely have been a viable approach.

We now draw comparative conclusions between the leaf lists of RTA and those of two-tree. The r-trees created by the two approaches are identical. Therefore, the r-tree leaf lists are identical. The difference between the two approaches is simply the number of i-trees created. Two-tree creates one long i-tree leaf list. RTA creates multiple shorter i-tree leaf lists. Since both approaches process the same r, i pairs, it is tempting to conclude that the concatenation of the smaller leaf lists of RTA would result in the longer leaf list of two-trees. If this were true, the overhead of the two approaches would be almost identical. An examination of the distributed tree depicted in table 16 above, however, reveals the flaw in this supposition. There are overlaps between the leaves of several of the i-trees. Specifically, i-trees 1, 4, 5 all have an i = 13 leaf. This demonstrates that in RTA some of the leaves may occur more than once in the header.

Thus, the header of RTA will be larger than that of two-tree by the number of overlapping leaves.

The actual number of overlapping leaves in RTA is data dependent. If the number of overlapping leaves is low, we expect a header size similar to that of two-trees and well below that of Huffman. If the number of overlapping leaves in RTA is high, it is possible that the header could approach or even exceed that of standard Huffman. The actual header sizes found for the test suite files using RTA and standard Huffman encoding are shown below.

File	Selected n			
	8	12	16	20
kennedy.xls	253	1552	2899	26869
plabrn12 txt	98	988	1891	12346
icet10 txt	100	1443	2931	16068
asyoulik txt	86	1056	1817	10164
alice29 txt	91	1107	1965	10074
grammer lsp	93	493	652	1501
cp html	104	1109	2063	6351
fields c	106	821	1150	3104
xargs 1	91	601	802	1926
sum	250	2073	4058	10274
ptt5	179	961	3958	11429
lenna	255	4499	58125	358290

Table 9: Header Size for RTA

File	Selected n			
	8	12	16	20
kennedy.xls	276	1883	3342	33171
plabrn12.txt	98	1188	2203	15159
icet10.txt	99	1786	3456	19763
asyoulik.txt	83	1283	2111	12444
alice29.txt	90	1348	2289	12337
grammer.lsp	91	576	728	1746
cp.html	100	1357	2409	7715
fields.c	106	984	1311	3703
xargs.1	87	711	904	2260
sum	273	2608	4805	12557
ptt5	177	1145	4670	13971
lenna	278	5768	71159	447245

Table 10: Header Size for Standard Huffman Encoding

Tables 9 and 10 demonstrate that the RTA header is in fact typically smaller than the header of standard Huffman encoding at n = 8. Further, it is apparent that as n increases the gap in the header size difference increases in favor of RTA over Huffman. This fits well with the empirical data (see Appendix B) that shows RTA performing best at large values of n's. Thus, we have probably identified the key element responsible for the (limited) success of RTA – decreased header size. This is no surprise as headers are the likely place to economize over the provably optimal encoding

G. SUMMARY

We now summarize our conclusions from the previous section regarding RTA vs. standard Huffman encoding.

1. RTA compresses typical text data to within ±0.5% of Huffman encoding.
2. RTA is computationally more expensive than Huffman encoding, though not prohibitively so.

3. RTA is space (memory) inefficient to the point of being infeasible.
4. The non-header portion of a file compressed by RTA typically comes within 5% of the non-header portion of the same file compressed by standard Huffman encoding.
5. RTA has a more compact header than standard Huffman encoding. The difference between the headers increases as n increases. Thus, RTA performs best on large files with a high n.

V. INDEXED TREE APPROACH

A. INTRODUCTION

In this chapter, we attempt to improve upon RTA by causing the distributed tree to more closely approach an optimal Huffman tree. We also seek to further decrease the size of the header and to reduce the memory requirements of our earlier approach.

One way to improve the distributed tree created by RTA is to decrease the depth and breadth of the i-trees. It is tempting to conclude that this can be accomplished simply by increasing the total number of i-trees. Unfortunately, increasing the number of i-trees requires an increase in the length of each r-code, which could prove detrimental to the overall compression ratio. Thus, we consider a more conservative approach to improving the distributed tree of RTA.

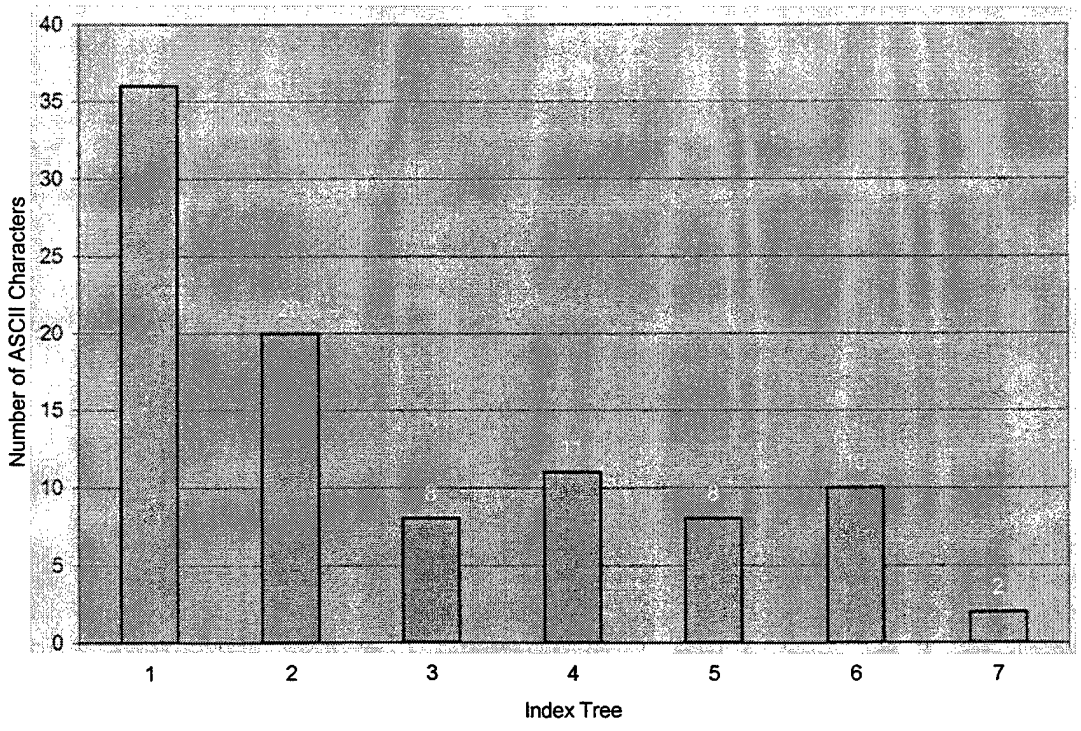


Chart 1: Dispersion of ASCII Characters by RTA (n = 8)

Table 21 summarizes the mapping between ASCII keyboard characters and the index trees of RTA (see Appendix B for details). The graph provides some indication of how good a job RTA is doing in dispersing i values to its forest of i-trees. It seems reasonable that a more even distribution between index trees would benefit the distributed tree of RTA. In fact, analysis of the actual index trees created by RTA when run on the test suit data shows that as n increases many trees are often left empty. This suggests that we can enhance the distributed tree of RTA by somehow doing a better job of evenly dispersing i values to the i-tree forest.

B. EXPLANATION

Our Indexed Tree Approach (ITA) completely discards necklace classes and rotations as a mechanism to create a Huffman forest. Instead, as we read an input symbol S we use the first few bits of S (iBits) as a straightforward index into our Huffman forest. The remaining bits of S (jBits) are then used as a leaf value for the tree indexed by iBits. We make the following clarifying remarks:

- S is the concatenation of iBits with jBits.
- $|iBits| + |jBits| = n$.
- The integer representation of the unsigned value iBits is denoted by [iBits].

ITA proceeds as follows: We first read n-bit symbols from a source. We then divide each symbol into its respective iBits and jBits components. We put the iBits into one Huffman tree, and distribute the jBits into a forest of Huffman trees based upon the value of their associated iBits components. Finally, we write the Huffman codes for each iBits, jBits pair to the output file in the order they were originally discovered.

Clearly, ITA is a derivative of RTA. ITA creates a similar Huffman forest, and constructs similar ordered pairs of Huffman codes. It is, however, much more flexible than RTA. In RTA, a given value n *determines* the length of the r- and i- components of the algorithm. In ITA, the sum of |iBits| and |jBits| *determines* the value of n. Thus, there is only one way for RTA to compress a given file at a given n, while there are many ways for ITA to compress the same file using the same n. For example, if n = 12, ITA can be executed with the following (|iBits|, |jBits|) pairs: (1, 11), (2, 10), (3, 9), (4, 8), (5, 7), (6, 6), (7, 5), (8, 4), (9,3), (10, 2) and (11, 1).

C. ALGORITHM

1. Build a static statistical model of the data as follows:
    a. Construct n + 1 empty frequency tables (indexed from 0 to n) where n>1 represents the length in bits of the symbols to read from the input stream.
    b. Read the next n-bit symbol S from the input stream.
    c. Divide S into its iBits and jBits components.
    d. If iBits does not exist in frequency table n, put iBits into frequency table n with frequency 1. Else, increment the frequency count of iBits in table n.
    e. If jBits does not exist in frequency table [iBits], put jBits into frequency table [iBits] with frequency 1. Else, increment the frequency count of jBits in table [iBits].
    f. While more symbols remain in the input stream, repeat step *a*.

2. Create a Huffman tree from each of the n + 1 frequency tables. The leaves of Huffman tree n represent iBits. The leaves of all other Huffman trees represent jBits.
3. Reset the input pointer to the beginning of the input stream.
4. Read the next n-bit symbol S from the input stream.
5. Divide S into its iBits and jBits components.
6. Write the Huffman code for iBits in Huffman tree n to the output stream.
7. Write the Huffman code for jBits in Huffman tree [iBits] to the output stream.
8. While more symbols remain in the input stream, repeat step 4.

D. INTERPRETATION OF RESULTS

We successfully implemented ITA in Java (see Appendix A) and collected empirical data (see Appendix B). In general, the compression performance of ITA is typically superior to that of standard Huffman encoding (and RTA) by a margin of one to three percent on text data. This difference is due to further decreases in the compressed file header size. ITA's performance is roughly equivalent to that of Huffman encoding on uncompressed bitmapped images. ITA is computationally more expensive than Huffman encoding, though like RTA, the difference is not apparent to the user. ITA does not suffer from the memory problems of RTA. In fact, it generally has a smaller memory footprint than does Huffman encoding.

Inspection of table 22 below (extracted from our empirical results, see appendix B) reveals that the performance gains made by ITA over Huffman and RTA are primarily attributable to further decreases in compressed file header size.

File	n	Optimal (|iBits|, |jBits|)	Huffman Header	RTA Header	ITA Header	Huffman Non-Header	RTA Non-Header	ITA Non-Header
kennedy.xls	8	(4,4)	276	253	191	462532	528207	516085
	12	(5,7)	1883	1552	1324	495537	502365	505657
	16	(7,9)	3342	2899	2476	410583	422265	421657
	20	(9,11)	33171	26869	20596	422785	426101	428105
plabrn12.txt	8	(4,4)	98	98	88	275585	288973	283622
	12	(4,8)	1188	988	910	306646	308099	306952
	16	(5 11)	2203	1891	1630	238776	241024	239739
	20	(9,11)	15159	12346	9652	255407	255701	256850
icet10.txt	8	(4,4)	99	100	85	250565	260360	255465
	12	(4,8)	1786	1443	1324	280698	282929	281309
	16	(5,11)	3456	2931	2499	218529	220143	218889
	20	(9,11)	19763	16068	12453	231209	231767	232403
asyoulik.txt	8	(4,4)	83	86	76	75806	78363	77650
	12	(5,7)	1283	1056	907	83237	83700	83528
	16	(7,9)	2111	1817	1469	64531	64969	64891
	20	(9,11)	12444	10164	8151	67550	67644	67834
alice29.txt	8	(4,4)	90	91	82	87688	91563	90150
	12	(4,8)	1348	1107	1020	98274	98839	98409
	16	(5,11)	2289	1965	1681	76120	77014	76522
	20	(9,11)	12337	10074	8076	80171	80324	80626
grammer.lsp	8	(4,4)	91	93	83	2170	2227	2205
	12	(5,7)	576	493	466	2289	2319	2284
	16	(7,9)	728	652	630	1703	1715	1718
	20	(6,14)	1746	1501	1410	1645	1656	1650
cp.html	8	(4,4)	100	104	89	16199	16389	16575
	12	(6,6)	1357	1109	927	17268	17422	17432
	16	(7,9)	2409	2063	1652	13337	13389	13419
	20	(8,12)	7715	6351	5390	12909	12974	12967
fields.c	8	(2,6)	106	106	90	7026	7104	7027
	12	(5,7)	984	821	701	7411	8391	8290
	16	(6,10)	1311	1150	972	5529	5538	5552
	20	(7,13)	3703	3104	2808	5408	5436	5429
xargs.1	8	(2,6)	87	91	81	2602	2662	2645
	12	(5,7)	711	601	528	2792	2809	2802
	16	(7,9)	904	802	697	2113	2127	2122
	20	(8,12)	2260	1926	1780	2000	2006	2009
sum	8	(2,6)	273	250	229	25645	26706	26238
	12	(5,7)	2608	2073	1735	24733	25221	24984
	16	(7,9)	4805	4058	3213	19977	20242	20167
	20	(8,12)	12557	10274	8681	19355	19503	19440
ptt5	8	(3,5)	177	179	160	106551	158044	158279
	12	(4,8)	1145	961	866	86993	119953	119939
	16	(6,10)	4670	3958	3227	76523	100685	100568
	20	(9,11)	13971	11429	8911	70124	88956	88886

Table 12: Header and Non-header Compression Results (in bytes)

In order to explain the decrease in header size from RTA to ITA we draw attention to the "optimal (|iBits|, |jBits|)" values displayed in column 3 of table 12. In most cases |iBits| and |jBits| are tightly centered on n/2. The reason for this goes back to the leaf lists discussed in chapter 4. The length and number of leaves in the leaf lists of a single iBits and jBits tree are given below.

	Leaf Length (bits)	Upper Bound on Number of Leafs
iBits tree	|iBits|	$2^{\|iBits\|}$
jBits tree	|jBits|	$2^{\|jBits\|}$

Table 13: Leaf Lists in a Single iBits and jBits Tree

An upper bound on the max leaf list size in bytes for a single iBits and jBits tree is found by summing the product of the leaf length and the number of leaves for each leaf list then dividing by 8. Table 14, below, shows how the upper bound for the size of the combined leaf lists of a single iBits and jBits tree change as i increases and j decreases.

n = 10 (|iBits|, |jBits|)	$(\|iBits\| * 2^{\|iBits\|} + \|jBits\| * 2^{\|jBits\|}) / 8$
(1, 9)	576
(2, 8)	257
(3, 7)	115
(4, 6)	56
(5, 5)	40
(6, 4)	56
(7, 3)	115
(8, 2)	257
(9, 1)	576

Table 14: Sum of Leaf List Upper Bounds for a single iBits and jBits Tree

Table 14 illustrates how the (|iBits|, |jBits|) = (n/2, n/2) is a minimum of it's $f(|iBits|, |jBits|) = (|iBits| * 2^{|iBits|} + |jBits| * 2^{|jBits|}) / 8$. Thus (|iBits|, |jBits|) = (n/2, n/2)

produces a minimal size header for a hypothetical two-tree ITA. Since the actual ITA uses many jBits trees we would expect some repeated leaves in the header (just as in RTA). Thus, while it is possible for the ITA header to exceed the shown values, the table suggests optimal |iBits|, |jBits| pairs that fit well with the empirical data.

Naturally, (|iBits|, |jBits|) = (n/2, n/2) need not always be the exact optimal value. Indeed, this decomposition is only optimal in about 15% of our test cases. The most common optimal (|iBits|, |jBits|) pairs for n = 8, 12, 16, and 20 are (4, 4), (5, 7), (7, 9), and (9, 11) respectively. Thus, there are other factors at work besides simply header overhead. The most critical factor in overall compression results for ITA is the degree of similarity between ITA's distributed tree and an optimal Huffman tree constructed with the same data. If the distributed tree performs poorly, any gains made by reduced header overhead are quickly lost.

As mentioned earlier, ITA is more memory efficient than standard Huffman encoding. The reasons for this are similar to those behind the smaller header – it is simply cheaper to build many small Huffman trees than it is to build a single large one.

Since our performance gains over ITA are due to smaller header size, we conclude that ITA is no better at dispersing jBits than RTA was at dispersing i-values. Thus, although we are able to enhance the overall compression performance of RTA, we are unable to improve its distributed tree.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSIONS

A. SUMMARY OF MAIN RESULTS

We have presented two lossless compression techniques. Both techniques use a static statistical data model. Our first approach (RTA) involves binary necklace classes and multiple Huffman trees.

1. RTA compresses typical text data to within ±0.5% of standard Huffman encoding.
2. RTA is computationally more expensive than Huffman encoding, though not prohibitively so.
3. RTA is space- (memory-) inefficient to the point of being infeasible.
4. The non-header portion of a file compressed by RTA typically comes within 5% of the non-header portion of the same file compressed by standard Huffman encoding.
5. RTA has a more compact header than standard Huffman encoding. The difference between the headers increases as n increases. Thus, RTA performs best on large files with a high n.

Our second lossless compression approach (ITA) uses the same basic model as RTA, but abandons binary necklaces in lieu of a simpler and more efficient tree indexing mechanism.

6. ITA compresses typical text data one to three percent better than standard Huffman encoding. Compression of uncompressed bitmapped images is roughly equivalent to that of Huffman encoding.
7. ITA is computationally more expensive than Huffman encoding, though not prohibitively so.
8. ITA requires less memory than standard Huffman encoding.
9. The non-header portion of a file compressed by ITA typically comes within 5% of the non-header portion of the same file compressed by standard Huffman encoding.
10. ITA has a more compact header than standard Huffman encoding. The difference between the headers tends toward a maximum at values centered on n/2 and becomes more pronounced as n increases.

B. FUTURE RESEARCH

This paper explores the use of binary necklace classes for lossless compression using a static statistical model. Our conclusions indicate that it is cheaper to pass header data for many small trees rather than a single large tree. Our approaches use one level of indirection (a single level of indirection equates to one thing pointing at many things) – a single tree pointing at a single forest. An interesting extension might be to have two or more levels of indirection. With two levels of indirection a single tree would point at a forest of trees, and each tree in the forest would then point at another forest. This

approach would greatly reduce the leaf length of each tree. The drawback is that the total number of trees would increase exponentially - thus greatly increasing the chance of overlapping leaves, something we found to be a drawback.

A mathematical topic closely related to necklace classes is that of Lyndon words. Lyndon words offer an opportunity to explore lossless compression using a dictionary-based model. Lyndon words exhibit some interesting properties, which may be applicable to data compression. Chapter 3 includes a short discussion on Lyndon words. Appendix A contains some preliminary java code, which generates an interesting subset of the Lyndon words.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A: JAVA CODE

```
package thesis.compression.multi_tree;


/**
 *  Allows C++ style assertions to be inserted in Java code for testing
and debugging.
 */
public class Assertion {
    public static boolean assertOn = true;

    private Assertion() {};    // no public constructor

    public static void assert(boolean validFlag)
    throws
    AssertionException
    {
        if (assertOn && !validFlag) {
            throw new AssertionException();
        }
    }

    public static void assert(boolean validFlag, String msg)
    throws
    AssertionException
    {
        if (assertOn && !validFlag) {
            throw new AssertionException(msg);
        }
    }
}


package thesis.compression.multi_tree;

public class AssertionException extends RuntimeException {
    public AssertionException() {
        super("AssertionException");
    }

    public AssertionException(String msg) {
        super(msg);
    }
}



package thesis.compression.multi_tree;

/**
 *  This class is an abstraction for a string of bits.  BitString
objects have length and
```

```
 *  bitPattern attributes.  This implementation limits the length of a
BitString to 32 bits.
 *  BitString objects are mutable.  This means that both the length and
bitPattern attributes
 *  of an instance can be changed by setter methods after the
instantiation of the object.
 */
public class BitString implements Comparable {

    public static final short MAXLEN;
    public static final int[] MASK;
    public static final int[] MASK_R;
    public static final int[] MASK_L;
    static {
        MAXLEN = 32;
        MASK = new int[MAXLEN];
        MASK_R = new int[MAXLEN];
        MASK_L = new int[MAXLEN];
        for (int i = 0; i < MAXLEN; i++) {
           MASK[i] = 1 << i;
           MASK_R[i] = -1 >>> 31 - i;
           MASK_L[i] = -1 << i;
        }
    }

    private short length;
    private int bitPattern;

    /**
     *  Returns one more than the number of bits to the right of the
most significant
     *  bit in bitPattern.  Alternatively, this may be thought of as
the minimum length
     *  required to capture all the significant bits of bitPattern.
     */
    public static short length(int bitPattern) {
        for (int i = 0; i < 32; i++) {
            if (bitPattern < 0) {
                return (short)(32 - i);
            }
            bitPattern <<= 1;
        }
        return 0;
    }

    /**
     *  Operates like System.arraycopy except on BitStrings instead of
arrays.
     *
     *  @throws IndexOutOfBoundsException if a copy operation would
exceed the bounds
     *          of either the source or destination BitString object.
     */
    public static void bitCopy(BitString src,
                               int srcPos,
                               BitString dst,
                               int dstPos,
```

```
                                    int numBits) throws
IndexOutOfBoundsException {

        int srcEnd = srcPos + numBits - 1;
        int dstEnd = dstPos + numBits - 1;

        if (srcPos < 0 || dstPos < 0 || numBits < 0 ||
            srcEnd >= src.length || dstEnd >= dst.length) {
            throw new IndexOutOfBoundsException();
        }

        // isolate the source bit field
        int s = src.bitPattern;
        int d = dst.bitPattern;
        int m = MASK_L[srcPos] & MASK_R[srcEnd];
        s &= m;

        // allign the mask and source bit field with the destination
bit field
        if (srcPos < dstPos) {
            int lshift = dstPos - srcPos;
            s <<= lshift;
            m <<= lshift;
        } else {
            int rshift = srcPos - dstPos;
            s >>>= rshift;
            m >>>= rshift;
        }

        // clear the dest bit field then AND in the source bit field
        dst.bitPattern = s | (d & (~m));
    }

    /**
     *  Returns the BitString object represented by str.
     *
     *  @throws  NumberFormatException just as Integer.parseInt(str, 2)
would.
     */
    public static BitString parseBitString(String str) throws
NumberFormatException {
        return new BitString(str.length(), Integer.parseInt(str, 2));
    }

    /*
     *  Creates a BitString[] to represent intArr. All BitStrings of
the returned array
     *  will be long enough to contain the greatest value in the int[].
     */
    public static BitString[] toBitStringArr(int[] intArr) {
        if (intArr == null) return null;

        // find greatest int value in the array
        int greatest = intArr[0];
        for (int i = 1; i < intArr.length; i++) {
            if (intArr[i] > greatest) {
                greatest = intArr[i];
```

```
            }
        }
        int bitsNeeded = (int)BitString.length(greatest);

        // create the BitString[]
        BitString[] retValue = new BitString[intArr.length];
        for (int i = 0; i < retValue.length; i++) {
            retValue[i] = new BitString(bitsNeeded, intArr[i]);
        }

        return retValue;
    }

    /**  Constructs a default BitString object of length = 0,
bitPattern = 0 */
    public BitString() {
        length = 0;
        bitPattern = 0;
    }

    /**
     *  Constructs a BitString object to represent bp
     *
     *  @throws  NumberFormatException exactly as Integer.parseInt(bp,
2)
     */
    public BitString(String bp) throws NumberFormatException {
        bitPattern = Integer.parseInt(bp, 2);
        length = (short)bp.length();
    }

    /**
     *  Constructs a BitString object of length len with the bitPattern
attribute
     *  represented by bp.
     *
     *  @throws  NumberFormatException exactly as Integer.parseInt(bp,
2)
     */
    public BitString(short len, String bp) throws NumberFormatException
{
        reInit(len, Integer.parseInt(bp, 2));
    }

    /** Constructs a BitString object of length len and bitPattern bp.
*/
    public BitString(short len, int bp) {
        reInit(len, bp);
    }

    /** Constructs a BitString object of length len and bitPattern 0.
*/
    public BitString(short len) {
        reInit(len, 0);
    }
```

```
    /** Constructs a BitString object of minimum length needed to
properly represent bp. */
    public BitString(int bp) {
        bitPattern = bp;
        length = length(bp);
    }

    /** Constructs a BitString object of mimimum length needed to
properly represent bp  */
    public BitString(BitString bp) {
        reInit(bp);
    }

    /**  allows package classes to create BitStrings without bounds
checking */
    BitString(int len, int bp) {
        bitPattern = bp;
        length = (short)len;
    }

    /** Returns true if the bit at bit position index is a 1.  */
    public final boolean bitAt(int index) throws
IndexOutOfBoundsException {
        boundsCheck(index);
        return (MASK[index] & bitPattern) != 0;
    }

    /** Sets the bit at bit position index to 1.  */
    public final void setBit(int index) throws
IndexOutOfBoundsException {
        boundsCheck(index);
        bitPattern |= MASK[index];
    }

    /** Clears the bit at bit position index  */
    public final void clearBit(int index) throws
IndexOutOfBoundsException {
        boundsCheck(index);
        bitPattern &= ~MASK[index];
    }

    /** Sets the bit at bit position index if value is true, else
clears the bit.  */
    public final void assignBit(int index, boolean value) throws
IndexOutOfBoundsException {
        boundsCheck(index);
        if (value == true) {
            bitPattern |= MASK[index];
        } else {
            bitPattern &= ~MASK[index];
        }
    }

    /** Returns the length of this BitString.  */
    public final short length() {
        return length;
    }
```

```
    /**
     *  Sets the length of this BitString to len.  This may result in
truncation if
     *  len < this.length.
     *
     *  @  LengthOutOfBoundsException if len < 0 or len >
BitString.MAXLEN.
     */
    public final void setLength(short len) throws
LengthOutOfBoundsException {
        reInit(len, bitPattern);
    }

    /** Returns this BitStrings bitPattern attribute.  */
    public final int bitPattern() {
        return bitPattern;
    }

    /**
     *  Sets this BitStrings bitPattern attribute to bp.  Note, if
this.length is not
     *  great enough bp may be truncated.
     */
    public final void setBitPattern(int bp) {
        reInit(length, bp);
    }

    /**
     *  Sets this BitStrings bitPattern to represent that of bp.  Note,
if this.length is
     *  not great enough bp may be truncated.
     *
     *  @throws  NumberFormatException exactly as Integer.parseInt(bp,
2).
     */
    public final void setBitPattern(String bp) throws
NumberFormatException {
        reInit(length, Integer.parseInt(bp, 2));
    }

    /** allows package classes to reInit BitStrings without bounds
checking */
    final void reInit(int len, int bp) {
        length = (short)len;
        bitPattern = bp;
    }

    /**
     *  Reinitializes this BitString to length len and bitPattern bp
     *
     *  @throws  NumberFormatException exactly as Integer.parseInt(bp,
2)
     */
    public final void reInit(short len, String bp) throws
NumberFormatException {
        reInit(len, Integer.parseInt(bp, 2));
```

```
    }

    /**
     *  Reinitializes this BitString to length len and bitPattern bp
     *
     *  @throws  LengthOutOfBoundsException if len < 0 OR len > MAXLEN.
     */
    public final void reInit(short len, int bp) throws
LengthOutOfBoundsException {
        if (len < 0 || len > MAXLEN) {
            throw new LengthOutOfBoundsException();
        } else if (len == 0) {
            length = 0;
            bitPattern = 0;
        } else {
            length = len;
            bitPattern = bp & MASK_R[length - 1];
        }
    }

    /** Reinitializes this BitString to bp  */
    public final void reInit(BitString bp) {
        length = bp.length;
        bitPattern = bp.bitPattern;
    }

    /** AND's this BitString with bp.  Only those bits that overlap
between this BitString
     *  and bp are affected
     */
    public final void and(BitString bp) {
        bitPattern &= (MASK_L[bp.length] | bp.bitPattern);
    }

    /** OR's this BitString with bp.  Only those bits that overlap
between this BitString
     *  and bp are affected
     */
    public final void or(BitString bp) {
        if (length == 0) return;
        bitPattern |= (bp.bitPattern & MASK_R[length - 1]);
    }

    /**
     *  Logically shifts the bits of this BitString one position to the
left.  If wrapOn
     *  is TRUE then the most significant bit will be shifted into the
least significant
     *  bit position.
     */
    public final int lShift(boolean wrapOn) {
        if (length == 0) {
            return 0;
        }

        int highBit = (MASK[length - 1] & bitPattern) != 0 ? 1 : 0;
        bitPattern <<= 1;
```

```
        bitPattern &= MASK_R[length - 1];

        if (wrapOn) {
            bitPattern |= highBit;
        }

        return highBit;
    }

    /**
     *  Logically shifts the bits of this BitString numBits %
this.length to the left.
     *  If wrapOn is TRUE then bits are shifted in from the right as
they are shifted
     *  out from the left.
     */
    public final void lShift(int numBits, boolean wrapOn) {
        if (numBits < 0) {
            rShift(numBits * -1, wrapOn);
        } else if (numBits * bitPattern == 0) {
            ;
        } else if (!wrapOn) {
            // lShift with true has bug
            bitPattern <<= numBits;
            bitPattern &= MASK_R[length - 1];
        } else {
            int nBits = numBits % length;
            int m = bitPattern & (MASK_R[length - 1] & MASK_L[length -
nBits]);
            bitPattern &= ~m;
            bitPattern <<= nBits;
            bitPattern |= (m >>> (length - nBits));
        }
    }

    /**
     *  Logically shifts the bits of this BitString one position to the
right.  If wrapOn
     *  is TRUE then the least significant bit will be shifted into the
most significant
     *  bit position.
     */
    public final int rShift(boolean wrapOn) {
        int lowBit = bitPattern & 1;
        bitPattern >>>= 1;

        if (wrapOn && lowBit == 1) {
            bitPattern |= MASK[length - 1];
        }

        return lowBit;
    }

    /**
     *  Logically shifts the bits of this BitString numBits %
this.length to the right.
```

```
     *  If wrapOn is TRUE then bits are shifted in from the left as
they are shifted
     *  out from the right.
     */
    public final void rShift(int numBits, boolean wrapOn) {
        if (numBits < 0) {
            lShift(numBits * -1, wrapOn);
        } else if (numBits * bitPattern == 0) {
            ;
        } else if (!wrapOn) {
            bitPattern >>>= numBits;
        } else {
            int nBits = numBits % length;
            int m = bitPattern & MASK_R[nBits - 1];
            bitPattern >>>= nBits;
            bitPattern |= (m << (length - nBits));
        }
    }

    /** Returns the position of the least significant 1 in this
BitString */
    public final int leastSig1() {
        int bp = bitPattern;
        int i = 0;
        while ((bp & 1) == 0) {
            bp >>>= 1;
            i++;
        }
        return i < MAXLEN ? i : -1;
    }

    /**
     *  Concatenates rVal to this BitString.
     *
     *  @throws LengthOutOfBoundsException if length + rVal.length >
BitString.MAXLEN.
     */
    public final void concat(BitString rVal) throws
LengthOutOfBoundsException {
        short concatLen = (short)(length + rVal.length);
        if (concatLen > MAXLEN) {
            throw new LengthOutOfBoundsException(
                 "Concatenated result would exceed " + MAXLEN);
        }
        bitPattern <<= rVal.length;
        bitPattern |= rVal.bitPattern;
        length = concatLen;
    }

    public int hashCode() {
        return bitPattern;
    }

    public String toString() {
        if (length == 0) return "[0]";
        char[] out = new char[length];
        for (int i = 0; i < length; i++) {
```

```
            out[length - 1 - i] = bitAt(i) ? '1' : '0';
        }
        return String.valueOf(out);
    }

    public final int compareTo(BitString bs) {
        if (length != bs.length) {
            // the longer BitString is always considered larger
            return length - bs.length;
        } else if ((bitPattern | bs.bitPattern) < 0) {
            // two MAXLEN BitStrings, one or both have their high bits
set
            if (bitPattern < 0 && bs.bitPattern >= 0)  {
                // this BitString negative, rhs non-negative
                return 1;
            } else if (bitPattern >= 0 && bs.bitPattern < 0) {
                // this BitString non-negative, rhs negative
                return -1;
            } else {
                // both negative
                return (bitPattern & Integer.MAX_VALUE) -
                       (bs.bitPattern & Integer.MAX_VALUE);
            }
        } else {
           //  two equal BitStrings w/o their high bits set
           return bitPattern - bs.bitPattern;
        }
    }

    public final int compareTo(Object obj) {
        return compareTo((BitString)obj);
    }

    public final boolean equals(Object obj) {
        if ((obj != null) && (obj instanceof BitString)) {
            BitString bs = (BitString)obj;
            return (length == bs.length) && (bitPattern ==
bs.bitPattern);
        }
        return false;
    }

    private final void boundsCheck(int index) throws
IndexOutOfBoundsException {
        if (index < 0 || index >= length) {
            throw new IndexOutOfBoundsException();
        }
    }
}



package thesis.compression.multi_tree;


import java.io.*;
```

```
import java.util.*;

/**
 *  Objects of this class are used to read data from a file.  Data is
read in chunks
 *  from 1 to BitString.MAXLEN bits.
 */
public class BitStringReader implements ByteMask {

    private static final int BUF_SIZE  = 81;  //92
    private static final int DEFAULT_N = 8;

    private byte[] buf;                  // holds bytes read in from
inStream
    private int endOfBuf;                // first unusable byte in buf
    private int oldEndOfBuf;             // remembers endOfBuf in case
reset() needs it
    private int bufFilled;               // # times this buf was filled
(and overwritten)
    private long peekSizeAdj;            // corrects 'bits read'
inaccuracy induced by peekBitString()
    private String filename;             // name of the file to be
opened / reopened
    private FileInputStream inStream;    // underlying data stream
    private int n;                       // default BitString length to
read
    private int bytePos;                 // byte marker within current
buf
    private int bitPos;                  // bit marker within current
byte
    private BitString scratch;           // helps avoid unnecessary
object creation
    private long numDecoded;             // # Huffman codes decoded with
this reader
    private long decodeLimit;            // max # Huffman codes to
decode with this reader

    /**
     *  Constructs an instance that reads from file filename and uses n
as the
     *  default number of bits to read for calls to readBitString().
     */
    public BitStringReader(String filename, int n)
    throws
    FileNotFoundException,
    IOException,
    LengthOutOfBoundsException
    {
        buf = new byte[BUF_SIZE];
        inStream = new FileInputStream(filename);
        this.filename = filename;
        setN(n);
        endOfBuf = inStream.read(buf);
        oldEndOfBuf = endOfBuf;
        bufFilled = bytePos = 0;
        bitPos = 7;
        scratch = new BitString();
```

```
        numDecoded = peekSizeAdj = 0;
        decodeLimit = Long.MAX_VALUE;
    }

    /**
     *  Constructs an instance that reads from file filename and uses
DEFAULT_N as the
     *  default number of bits to read for calls to readBitString().
     */
    public BitStringReader(String filename)
    throws
    FileNotFoundException,
    IOException,
    LengthOutOfBoundsException
    {
        this(filename, DEFAULT_N);
    }

    /**
     *  Changes the number of bits read by a call to readBitString()
from the current
     *  value to n.
     *
     *  @throws  LengthOutOfBoundsException if n < 0 or n >
BitString.MAXLEN.
     */
    public final void setN(int n) throws LengthOutOfBoundsException {
        if (n < 0 || n > BitString.MAXLEN) {
            throw new LengthOutOfBoundsException();
        }

        this.n = n;
    }

    /** Sets the number of BitStrings readable from a file using
decodeBitString(). */
    public final void setDecodeLimit(long limit) {
        decodeLimit = limit;
    }

    /** Returns the number of bits read by this BitStringReader. */
    public final long bitsRead() {
        return (long)bufFilled * (long)BUF_SIZE * 8L + (long)bytePos *
8L +
                7L - (long)bitPos + (long)peekSizeAdj * 8L;
    }

    /** Returns the filename of the file being read by this
BitStringReader. */
    public final String filename() {
        return filename;
    }

    /**
     *  Returns a BitString that represents the next n bits of the file
being
```

```
     * read by this BitStringReader.  n is an already set instance
variable of
     * this BitStringReader.  Thus, this is the default read operation
which is useful
     * if you need to read fixed-length bit strings from a source
file.  Note: if there
     * are only x bits left in the file where x < n then a BitString
of length x will be
     * returned.  Thus, a returned BitString of length < n is a signal
that the EOF has
     * been reached.  Further calls to this method after EOF will
return BitStrings of
     * length 0.
     */
    public final BitString readBitString() throws IOException {
        return readBitString(n);
    }

    /**
     * Returns a BitString that represents the next len bits of the
file being
     * read by this BitStringReader.  Note: if there are only x bits
left in the file
     * where x < len then a BitString of length x will be returned.
Thus, a returned
     * BitString of length < len is a signal that the EOF has been
reached.  Further calls
     * to this method after EOF will return BitStrings of length 0.
     */
    public BitString readBitString(int len)
    throws
    IOException,
    LengthOutOfBoundsException
    {
        int bitsNeeded = len;
        BitString temp = scratch;
        BitString bs = new BitString((short)bitsNeeded);

        while (bitsNeeded > 0) {
            if (bytePos == endOfBuf) {
                // at the end of the current buffer
                endOfBuf = inStream.read(buf);
                if (endOfBuf == -1) {
                    endOfBuf = bytePos;
                    bs.reInit(len - bitsNeeded, bs.bitPattern());
                    return bs;
                } else {
                    bufFilled++;
                    bytePos = 0;
                }
            } else if (bitsNeeded > bitPos) {
                // need all of the current byte
                int pieceLen = bitPos + 1;
                bs.lShift(pieceLen, false);
                temp.reInit(pieceLen, buf[bytePos] & MASK_R[bitPos]);
                bs.or(temp);
                bitsNeeded -= pieceLen;
```

```
                bytePos++;
                bitPos = 7;
            } else {
                // need only some of the current byte
                bs.lShift(bitsNeeded, false);
                int t = (buf[bytePos] & MASK_R[bitPos]) >>> (bitPos -
bitsNeeded + 1);
                temp.reInit(bitsNeeded, t);
                bs.or(temp);
                bitPos -= bitsNeeded;
                bitsNeeded = 0;
            }
        }
        return bs;
    }

    /**
     *  This method is identical in functionality to readBitString(len)
except that the
     *  the underlying state of this BitStringReader remains unchanged
after each call.
     *  Thus, multiple calls to this method are guaranteed to return
the same result
     *  so long as no intermediate calls to readBitString or
decodeBitString are made.
     *  This method is useful to determine what is coming up in the
input stream without
     *  actually advancing the input stream pointer.
     */
    public BitString peekBitString(int len)
    throws
    IOException,
    LengthOutOfBoundsException
    {
        int bitsNeeded = len;
        BitString temp = scratch;
        BitString bs = new BitString((short)bitsNeeded);
        int bitPos = this.bitPos;
        int bytePos = this.bytePos;
        int endOfBuf = this.endOfBuf;

        while (bitsNeeded > 0) {
            if (bytePos == endOfBuf) {
                // at the end of the current buffer
                int bytesFromOldBuf = this.endOfBuf - this.bytePos;
                System.arraycopy(buf, this.bytePos, buf, 0,
bytesFromOldBuf);
                endOfBuf = inStream.read(buf, bytesFromOldBuf, BUF_SIZE
- bytesFromOldBuf) +
                            bytesFromOldBuf;
                bufFilled++;
                peekSizeAdj -= (BUF_SIZE - this.bytePos);

                if (endOfBuf < bytesFromOldBuf) {
                    // no more bytes to read from source file
                    this.endOfBuf = bytesFromOldBuf;
                    this.bytePos = 0;
```

```
                    bs.reInit(len - bitsNeeded, bs.bitPattern());
                    return bs;
                } else {
                    //
                    this.endOfBuf = endOfBuf;
                    this.bytePos = 0;
                    bytePos = bytesFromOldBuf;
                }
            } else if (bitsNeeded > bitPos) {
                // need all of the current byte
                int pieceLen = bitPos + 1;
                bs.lShift(pieceLen, false);
                temp.reInit(pieceLen, buf[bytePos] & MASK_R[bitPos]);
                bs.or(temp);
                bitsNeeded -= pieceLen;
                bytePos++;
                bitPos = 7;
            } else {
                // need only some of the current byte
                bs.lShift(bitsNeeded, false);
                int t = (buf[bytePos] & MASK_R[bitPos]) >>> (bitPos -
bitsNeeded + 1);
                temp.reInit(bitsNeeded, t);
                bs.or(temp);
                bitPos -= bitsNeeded;
                bitsNeeded = 0;
            }
        }
        return bs;
    }

    /**
     * Reads lenOfShortestHufCode bits from the input stream and
checks to see if this
     * key is found in decodingMap.  If it is then a BitString
representing the value
     * mapped to by the key is returned.  Else, another bit is
concatenated to the
     * current key and another lookup in decodingMap is performed.  If
the new key
     * is found in decoding map then a BitString representing the
value mapped to by
     * the new key is returned.  This process continues untill a
BitString is returned,
     * the key exceeds BitString.MAXLEN (which throws a
LengthOutOfBounds Exception), or
     * there are no more bits to read from the input stream (which
throws an
     * AssertionException).
     */
    public final BitString decodeBitString(HashMap decodingMap,
                                           int lenOfShortestHufCode)
    throws
    IOException
    {
        if (numDecoded == decodeLimit) {
            return new BitString();
```

```
        }
        BitString hufCode = readBitString(lenOfShortestHufCode);
        BitString aBit;
        while (!decodingMap.containsKey(hufCode)) {
            aBit = readBitString(1);
            Assertion.assert(aBit.length() == 1);
            hufCode.concat(aBit);
        }
        numDecoded++;

        // returning the VALUE that hufCode maps to in the decoding map
(its class index)
        // reusing hufCode for return value to avoid object creation
        hufCode.reInit((BitString)decodingMap.get(hufCode));
        return hufCode;
    }

    // PRE: 0 < nBound <= BitString.MAXLEN
    // POST:  returns either,
    //          1)  a Lyndon word of length <= nBound
    //          2)  a contiguous string of 0's of length <= nBound
    //          3)  a contiguous string of 1's of length == nBound
    //          4)  any BitString of length <= nBound iff its the tail
of the source file
    //          5)  a BitString of length 0 iff no bits remain in the
source file
    public BitString readLyndonWord(int nBound)
    throws
    IOException
    {
        BitString sourceBits = peekBitString(nBound);
        if (sourceBits.length() < nBound) {
            // remainder bits, case 4 or 5
            return readBitString(sourceBits.length());
        }

        int returnLen = 0;
        int lenSoFar = 1;
        int aBit = sourceBits.lShift(false);

        if (aBit == 0) {
            // a leading 0, case 2
            aBit = sourceBits.lShift(false);
            while (aBit == 0 && lenSoFar < nBound) {
                lenSoFar++;
                aBit = sourceBits.lShift(false);
            }
            returnLen = lenSoFar;
        } else {
            // a leading 1, case 1 or 3
            aBit = sourceBits.lShift(false);
            while (aBit == 1 && lenSoFar < nBound) {
                lenSoFar++;
                aBit = sourceBits.lShift(false);
            }

            if (lenSoFar == nBound) {
```

```
                // case 3;
                returnLen = lenSoFar;
            } else {
                // case 1
                int numLeadIs = lenSoFar++;
                returnLen = lenSoFar;
                int numNonLeadIs = 0;
                aBit = sourceBits.lShift(false);
                while (numNonLeadIs < numLeadIs && lenSoFar < nBound)
{
                    lenSoFar++;
                    if (aBit == 0) {
                        returnLen = lenSoFar;
                        numNonLeadIs = 0;
                    } else {
                        numNonLeadIs++;
                    }
                    aBit = sourceBits.lShift(false);
                }
            }
        }

        return readBitString(returnLen);
    }

    /** Resets the input stream of this BitStringReader to the first
bit in the file. */
    public void reset() throws IOException {
        if (bufFilled > 0) {
            inStream.close();
            inStream = null;
            System.runFinalization();
            System.gc();
            inStream = new FileInputStream(filename);
            endOfBuf = oldEndOfBuf = inStream.read(buf);
            bufFilled = 0;
            peekSizeAdj = 0;

        } else {
            endOfBuf = oldEndOfBuf;
        }
        bytePos = 0;
        bitPos = 7;
        numDecoded = 0;
    }

    /**
     *  Closes this BitStringReader.
     */
    public void close() throws IOException {
        inStream.close();
    }
}
```

```
package thesis.compression.multi_tree;


import java.io.*;
import java.util.*;

/**
 *  Objects of this class are used to write variable length BitStrings
to a file.
 *  Upon closing the BitStringWriter any bits beyond the last byte
boundary will be
 *  padded with enough 0 bits as to make the total number of bits
written to the
 *  target file divisible by 8.
 */
public class BitStringWriter implements ByteMask {

    private static final int BUF_SIZE = 8192;
    private byte[] buf;
    private FileOutputStream outStream;
    private int bufFilled;                        // # times buf was
filled
    private int bytePos;
    private int bitPos;
    private BitString scratch;

    /** Constructs a BitStringWriter object to write to file 'file'. */
    public BitStringWriter(String file) throws FileNotFoundException {
        buf = new byte[BUF_SIZE];
        outStream = new FileOutputStream(file);
        bufFilled = 0;
        bytePos = 0;
        bitPos = 7;
        scratch = new BitString();
    }

    /** Returns the number of bits written by this BitStringWriter. */
    public long bitsWrote() {
        return (long)bufFilled * (long)BUF_SIZE * 8L + (long)bytePos *
8L + 7L - (long)bitPos;
    }

    /** Writes BitString bStr to the output stream */
    public void writeBitString(BitString bStr) throws IOException {
        BitString bs = scratch;
        bs.reInit(bStr);
        while (bs.length() > 0) {
            if (bytePos == BUF_SIZE) {
            // buffer is full
                outStream.write(buf);
                Arrays.fill(buf, (byte)0);
                bytePos = 0;
                bufFilled++;
            } else if (bs.length() > bitPos) {
            // need all of the current byte
                int bitsAvail = bitPos + 1;
```

```
                byte b = (byte)(bs.bitPattern() >>> bs.length() -
bitsAvail);
                buf[bytePos] |= b;
                bs.setLength((short)(bs.length() - bitsAvail));
                bytePos++;
                bitPos = 7;
            } else {
            // need only some of the current byte
                byte b = (byte)(bs.bitPattern() << bitPos - bs.length()
+ 1);
                buf[bytePos] |= b;
                bitPos -= bs.length();
                bs.setLength((short)0);
            }
        }
    }

    /**
     *  Convenience method that writes all the BitStrings of
BitString[] bsArr to
     *  the output stream in index order
     */
    public void writeBitString(BitString[] bsArr) throws IOException {
        for (int i = 0; i < bsArr.length; i++) {
            writeBitString(bsArr[i]);
        }
    }

    /**
     *  Closes this BitStringWriter.
     */
    public void close() throws IOException {
        outStream.write(buf, 0, bytePos + (bitPos == 7 ? 0 : 1));
        outStream.flush();
        outStream.close();
        outStream = null;
        System.runFinalization();
        System.gc();
    }
}
```

```
package thesis.compression.multi_tree;


public interface ByteMask {

    public static final int[] MASK_R =  {0x00000001, 0x00000003,
0x00000007, 0x0000000F,
                                         0x0000001F, 0x0000003F,
0x0000007F, 0x000000FF};

    public static final int[] MASK_L =  {0x000000FF, 0x000000FE,
0x000000FC, 0x000000F8,
```

```
                                                0x000000F0, 0x000000E0,
0x000000C0, 0x00000080};
}



package thesis.compression.huffman;

import java.util.*;
import java.io.*;
import thesis.compression.multi_tree.*;

/**
 *  An instance of this class compresses a sourceFile using standard
Huffman encoding
 *  and n-bit input symbols.
 */
public class HuffmanCompressionManager {

    HashMap freqTable;
    BitStringReader sourceFile;
    BitStringWriter targetFile;
    int n;
    BitString remainder;
    BitString offset;


    public static void main(String[] args) {

        String usage = new String("USAGE:  java
HuffmanCompressionManager " +
            "<sourceFilename> <targetFilename> <n> <offset>");

        try {
            if (args.length != 4) {
                throw new ArrayIndexOutOfBoundsException();
            }
            int n = Integer.parseInt(args[2]);
            int offset = Integer.parseInt(args[3]);
            BitStringReader sourceFile = new BitStringReader(args[0],
n);
            BitStringWriter targetFile = new BitStringWriter(args[1]);
            HuffmanCompressionManager hcm =
                new HuffmanCompressionManager(sourceFile, targetFile,
n, offset);
            hcm.compress();
        } catch (NumberFormatException arg3Or4NotInt) {
            System.out.println(usage);
        } catch (ArrayIndexOutOfBoundsException wrongNumArgs) {
            System.out.println(usage);
        } catch (LengthOutOfBoundsException paramsOutOfRange) {
            System.out.println("1 < n <= " + BitString.MAXLEN + "  AND
0 <= offset < n");
        } catch (Exception e) {
            e.printStackTrace(new PrintStream(System.out));
        } finally {
```

```
            System.exit(1);
        }
    }

    public HuffmanCompressionManager(BitStringReader sourceFile,
                                     BitStringWriter targetFile,
                                     int n,
                                     int offsetLen)
    throws
    IOException,
    LengthOutOfBoundsException
    {
        // bounds check
        if (n < 2 || n > BitString.MAXLEN || offsetLen < 0 || offsetLen
>= n) {
            throw new LengthOutOfBoundsException();
        }

        // initialize instance variables
        this.n = n;
        sourceFile.setN(n);
        this.sourceFile = sourceFile;
        this.targetFile = targetFile;
        offset = sourceFile.readBitString(offsetLen);
        remainder = new BitString();
        freqTable = new HashMap();
    }

    public void compress()
    throws
    IOException
    {
        buildFreqTable(sourceFile, freqTable, n, remainder);
        compressSourceFile(targetFile, sourceFile, freqTable, n,
remainder, offset);
    }

    public final long sizeAfter() {
        return targetFile.bitsWrote();
    }

    protected void buildFreqTable( /* IN  */ BitStringReader
sourceFile,
                                   /* OUT */ HashMap freqTable,
                                   /* IN  */ int n,
                                   /* OUT */ BitString remainder)
    throws
    IOException
    {
        BitString bs = sourceFile.readBitString();
        while (bs.length() == n) {
            if (freqTable.containsKey(bs)) {
                ((VonNoymanNode)(freqTable.get(bs))).incrFreq();
            } else {
                freqTable.put(bs, new VonNoymanNode(bs));
            }
            bs = sourceFile.readBitString();
```

```
        }
        remainder.reInit(bs);
    }

    protected void compressSourceFile(/* OUT    */ BitStringWriter
targetFile,
                                      /* IN     */ BitStringReader
sourceFile,
                                      /* IN/OUT */ HashMap freqTable,
                                      /* IN     */ int n,
                                      /* IN     */ BitString remainder,
                                      /* IN     */ BitString offset)
    throws
    IOException
    {
        // write compression header
        writeCompressionHeader(targetFile, sourceFile, n, freqTable,
remainder, offset);

        HashMap encodingMap = freqTable;
        sourceFile.reset();
        sourceFile.readBitString(offset.length());  // throw away
offset

        // write sourceFile to targetFile in compressed form
        BitString bs = sourceFile.readBitString();
        while (bs.length() == n) {
            bs = (BitString)encodingMap.get(bs);
            targetFile.writeBitString(bs);
            bs = sourceFile.readBitString();
        }
        Assertion.assert(remainder.equals(bs));
        targetFile.close();
    }

    protected void writeCompressionHeader(/* OUT    */ BitStringWriter
targetFile,
                                          /* IN     */ BitStringReader
sourceFile,
                                          /* IN     */ int n,
                                          /* IN/OUT */ HashMap
freqTable,
                                          /* IN     */ BitString
remainder,
                                          /* IN     */ BitString
offset)
    throws
    IOException
    {
        // first entries in header of target file
        targetFile.writeBitString(new BitString((short)5, n));
        targetFile.writeBitString(new BitString((short)5,
offset.length()));
        targetFile.writeBitString(offset);
        targetFile.writeBitString(new BitString((short)5,
remainder.length()));
        targetFile.writeBitString(remainder);
```

```
        // convert the HashMap from a freqency table to an encoding map
and write
        // header info for the Huffman tree
        processTree(freqTable, targetFile);

        // write numBitStringsRead as last entry in header
        long numBitStringsRead = (sourceFile.bitsRead() -
offset.length()) / n;
        Assertion.assert(numBitStringsRead <= Integer.MAX_VALUE);
        targetFile.writeBitString(new BitString((short)32,
(int)numBitStringsRead));
        System.out.println("****** HUFFMAN HEADER SIZE FOR " +
sourceFile.filename() +
            " with N = " + n + " is " + (targetFile.bitsWrote() / 8) +
" bytes ******");
    }

    protected void processTree(/*  IN/OUT  */  HashMap freqTable,
                               /*  OUT     */  BitStringWriter
targetFile)
    throws
    IOException
    {
        // if the freq table is empty write minimal header info and
return
        if (freqTable.isEmpty()) {
            targetFile.writeBitString(new BitString((short)5, 0));
            return;
        }

        // order all the VonNoymanNodes of this table based on
frequency
        TreeSet freqOrderedVNN = new TreeSet(freqTable.values());

        // build a frequency ordered bit string list for the
compression header
        BitString[] freqOrderedBitStringList = new
BitString[freqOrderedVNN.size()];
        Iterator it = freqOrderedVNN.iterator();
        int i = 0;
        while (it.hasNext()) {
            freqOrderedBitStringList[i++] =
((VonNoymanNode)it.next()).index;
        }

        // build leafs at level list for the compression header
        // NOTE:  this algorithm trashes its TreeSet parameter AND the
underlying
        // HashMap upon which it is based
        BitString[] leafsAtLevelList =
VonNoymanNode.vonNoymanAlgorithm(freqOrderedVNN);

        // write this tree's portion of the compression header
        BitString leafsPerLevel = new BitString((short)5,
leafsAtLevelList[0].length());
```

```
        BitString  numLevels = new BitString((short)5,
leafsAtLevelList.length);
        targetFile.writeBitString(leafsPerLevel);
        targetFile.writeBitString(numLevels);
        targetFile.writeBitString(leafsAtLevelList);
        targetFile.writeBitString(freqOrderedBitStringList);

        // build frequency ordered list of Huffman codes
        BitString[] freqOrderedHuffmanCodes =
VonNoymanNode.getHuffmanCodes(leafsAtLevelList);

        // reuse the trashed HashMap as a (class index --> Huffman
code) encoding map
        HashMap bitStringToHuffmanCodeMap = freqTable;
        for (int j = 0; j < freqOrderedBitStringList.length; j++) {
            bitStringToHuffmanCodeMap.put(
                freqOrderedBitStringList[j],
freqOrderedHuffmanCodes[j]);
        }
    }
}
```

```
package thesis.compression.huffman2;

import java.util.*;
import java.io.*;
import thesis.compression.multi_tree.*;
import thesis.compression.huffman.*;

/**
 *  This class is identical to HuffmanCompressionManager except that it
uses a slightly
 *  more efficient header encoding format.  The performance difference
between the two
 *  techniques is typically insignificant (<0.1%).
 */
public class HuffmanCompressionManager2 extends
HuffmanCompressionManager {

    public static void main(String[] args) {

        String usage = new String("USAGE:  java
HuffmanCompressionManager2 " +
            "<sourceFilename> <targetFilename> <n> <offset>");

        try {
            if (args.length != 4) {
                throw new ArrayIndexOutOfBoundsException();
            }
            int n = Integer.parseInt(args[2]);
            int offset = Integer.parseInt(args[3]);
            BitStringReader sourceFile = new BitStringReader(args[0],
n);
            BitStringWriter targetFile = new BitStringWriter(args[1]);
```

```
            HuffmanCompressionManager2 hcm2 =
                new HuffmanCompressionManager2(sourceFile, targetFile,
n, offset);
            hcm2.compress();
        } catch (NumberFormatException arg3Or4NotInt) {
            System.out.println(usage);
        } catch (ArrayIndexOutOfBoundsException wrongNumArgs) {
            System.out.println(usage);
        } catch (LengthOutOfBoundsException paramsOutOfRange) {
            System.out.println("1 < n <= " + BitString.MAXLEN + "  AND
0 <= offset < n");
        } catch (Exception e) {
            e.printStackTrace(new PrintStream(System.out));
        } finally {
            System.exit(1);
        }
    }

    public HuffmanCompressionManager2(BitStringReader sourceFile,
                                      BitStringWriter targetFile,
                                      int n,
                                      int offsetLen)
    throws
    IOException,
    LengthOutOfBoundsException
    {
        super(sourceFile, targetFile, n, offsetLen);
    }

    protected void processTree(/*  IN/OUT  */  HashMap freqTable,
                               /*  OUT     */  BitStringWriter
targetFile)
    throws
    IOException
    {
        // if the freq table is empty write minimal header info and
return
        if (freqTable.isEmpty()) {
            targetFile.writeBitString(new BitString((short)5, 0));
            return;
        }

        // order all the VonNoymanNodes of this table based on
frequency
        TreeSet freqOrderedVNN = new TreeSet(freqTable.values());

        // build a frequency ordered jWord list for the compression
header
        BitString[] freqOrderedJWordList = new
BitString[freqOrderedVNN.size()];
        Iterator it = freqOrderedVNN.iterator();
        int i = 0;
        while (it.hasNext()) {
            freqOrderedJWordList[i++] =
((VonNoymanNode)it.next()).index;
        }
```

```
        // build and trim leafs at level list for the compression
header
        // NOTE:  the VonNoyman algorithm trashes its TreeSet parameter
AND
        // the underlying HashMap upon which it is based)
        BitString[] leafsAtLevelList =
VonNoymanNode.vonNoymanAlgorithm(freqOrderedVNN);
        BitString[] trimedLeafsAtLevelList =
trimEmptyLeadLevels(leafsAtLevelList);
        int firstLevel = leafsAtLevelList.length -
trimedLeafsAtLevelList.length;

        // write this tree's portion of the compression header
        BitString numNonEmptyLevels = new BitString((short)5,
trimedLeafsAtLevelList.length);
        BitString firstNonEmptyLevel = new BitString((short)5,
firstLevel);
        BitString leafsPerLevel = new BitString((short)5,
trimedLeafsAtLevelList[0].length());
        targetFile.writeBitString(numNonEmptyLevels);
        targetFile.writeBitString(firstNonEmptyLevel);
        targetFile.writeBitString(leafsPerLevel);
        targetFile.writeBitString(trimedLeafsAtLevelList);
        targetFile.writeBitString(freqOrderedJWordList);

        // build frequency ordered list of Huffman codes
        BitString[] freqOrderedHuffmanCodes =
VonNoymanNode.getHuffmanCodes(leafsAtLevelList);
        Assertion.assert(freqOrderedJWordList.length ==
freqOrderedHuffmanCodes.length);

        // reuse the trashed HashMap as a (j-word --> Huffman code)
encoding map
        HashMap JWordToHuffmanCodeMap = freqTable;
        for (int k = 0; k < freqOrderedJWordList.length; k++) {
            JWordToHuffmanCodeMap.put(freqOrderedJWordList[k],
freqOrderedHuffmanCodes[k]);
        }
    }

    private final BitString[] trimEmptyLeadLevels(BitString[]
leafsAtLevelList) {
        int numNonEmptyLevels = leafsAtLevelList.length;
        int i = 0;
        while (leafsAtLevelList[i].bitPattern() == 0) {
            numNonEmptyLevels--;
            i++;
        }

        BitString[] newLeafsAtLevelList = new
BitString[numNonEmptyLevels];
        System.arraycopy(leafsAtLevelList, i, newLeafsAtLevelList, 0,
numNonEmptyLevels);
        return newLeafsAtLevelList;
    }
```

```
}
```

```
package thesis.compression.huffman;

import java.util.*;
import java.io.*;
import org.omg.CORBA.IntHolder;
import thesis.compression.multi_tree.*;

/**
 *  An instance of this class decompresses a file compressed by an
instance of
 *  HuffmanCompressionManager.
 */
public class HuffmanDecompressionManager {

    private BitStringWriter targetFile;
    private BitStringReader sourceFile;
    private HashMap decodingMap;
    private int n;
    private BitString remainder;
    private BitString offset;


    public static void main(String[] args) {

        String usage = new String("USAGE:  java
HuffmanDecompressionManager " +
            "<sourceFilename> <targetFilename>");

        try {
            if (args.length != 2) {
                throw new ArrayIndexOutOfBoundsException();
            }
            BitStringReader sourceFile = new BitStringReader(args[0]);
            BitStringWriter targetFile = new BitStringWriter(args[1]);
            new HuffmanDecompressionManager(sourceFile, targetFile);
        } catch (ArrayIndexOutOfBoundsException wrongNumArgs) {
            System.out.println(usage);
        } catch (Exception e) {
            e.printStackTrace(new PrintStream(System.out));
        } finally {
            System.exit(1);
        }
    }

    public HuffmanDecompressionManager(BitStringReader sourceFile,
                                       BitStringWriter targetFile)
    throws
    IOException
    {
        this.sourceFile = sourceFile;
        this.targetFile = targetFile;
```

```
        n = (sourceFile.readBitString(5)).bitPattern();
        if (n == 0) {
            n = 32;
        }
        int lenOffset = (sourceFile.readBitString(5)).bitPattern();
        offset = sourceFile.readBitString(lenOffset);
        int lenRem = (sourceFile.readBitString(5)).bitPattern();
        remainder = sourceFile.readBitString(lenRem);
        decodingMap = new HashMap();
        decompressSourceFile(targetFile, sourceFile, n, decodingMap,
remainder, offset);
    }

    protected void decompressSourceFile(/* OUT */  BitStringWriter
targetFile,
                                        /* IN  */  BitStringReader
sourceFile,
                                        /* IN  */  int n,
                                        /* OUT */  HashMap decodingMap,
                                        /* IN  */  BitString remainder,
                                        /* IN  */  BitString offset)
    throws
    IOException
    {
        targetFile.writeBitString(offset);

        // build decoding map (Huffman code ==> original bit string)
        // remember length of shortest Huffman code
        IntHolder lengthOfShortHufCode = new IntHolder();
        buildDecodingMap(sourceFile, n, decodingMap,
lengthOfShortHufCode);
        int lenOfShortHufCode = lengthOfShortHufCode.value;

        long numBitStringsCompressed =
(sourceFile.readBitString(32)).bitPattern();
        sourceFile.setDecodeLimit(numBitStringsCompressed);
        sourceFile.setN(n);

        BitString bs = sourceFile.decodeBitString(decodingMap,
lenOfShortHufCode);
        while (bs.length() > 0) {
            targetFile.writeBitString(bs);
            bs = sourceFile.decodeBitString(decodingMap,
lenOfShortHufCode);
        }

        sourceFile.close();
        targetFile.writeBitString(remainder);
        targetFile.close();
    }


    protected void buildDecodingMap(/* IN  */ BitStringReader
sourceFile,
                                    /* IN  */ int n,
                                    /* OUT */ HashMap decodingMap,
```

```
                                        /* OUT */ IntHolder
lenOfShortCodeInTable)
    throws
    IOException
    {
        // build leafsAtLevelList for getHuffmanCodes method
        int lenLevel = (sourceFile.readBitString(5)).bitPattern();
        int numLevels = (sourceFile.readBitString(5)).bitPattern();
        BitString[] leafsAtLevelList = new BitString[numLevels];
        for (int i = 0; i < numLevels; i++) {
            leafsAtLevelList[i] = sourceFile.readBitString(lenLevel);
        }

        // getHuffmanCodes
        BitString[] freqOrderedHuffmanCodes =
VonNoymanNode.getHuffmanCodes(leafsAtLevelList);

        // fill decodingMap with (Huffman code --> unencoded bit
string) mapping
        int numLeafs = freqOrderedHuffmanCodes.length;
        for (int i = 0; i < numLeafs; i++) {
            decodingMap.put(freqOrderedHuffmanCodes[i],
sourceFile.readBitString(n));
        }

        // find length of shortest Huffman code
        for (int i = 0; i < numLevels; i++) {
            if (leafsAtLevelList[i].bitPattern() != 0) {
                lenOfShortCodeInTable.value = i;
                return;
            }
        }
    }
}
```

```
package thesis.compression.huffman2;

import java.util.*;
import java.io.*;
import org.omg.CORBA.IntHolder;
import thesis.compression.multi_tree.*;
import thesis.compression.huffman.*;

/**
 *  Decompresses files compressed using HuffmanCompressionManager2.
 */
public class HuffmanDecompressionManager2 extends
HuffmanDecompressionManager {

    public static void main(String[] args) {
```

```
        String usage = new String("USAGE:  java
HuffmanDecompressionManager2 " +
            "<sourceFilename> <targetFilename>");

        try {
            if (args.length != 2) {
                throw new ArrayIndexOutOfBoundsException();
            }
            BitStringReader sourceFile = new BitStringReader(args[0]);
            BitStringWriter targetFile = new BitStringWriter(args[1]);
            new HuffmanDecompressionManager2(sourceFile, targetFile);
        } catch (ArrayIndexOutOfBoundsException wrongNumArgs) {
            System.out.println(usage);
        } catch (Exception e) {
            e.printStackTrace(new PrintStream(System.out));
        } finally {
            System.exit(1);
        }
    }

    public HuffmanDecompressionManager2(BitStringReader sourceFile,
                                        BitStringWriter targetFile)
    throws
    IOException
    {
        super(sourceFile, targetFile);
    }

    protected void buildDecodingMap(/* IN  */ BitStringReader
sourceFile,
                                    /* IN  */ int n,
                                    /* OUT */ HashMap decodingMap,
                                    /* OUT */ IntHolder
lenOfShortCodeInTable)
    throws
    IOException
    {
        int numNonEmptyLevels =
(sourceFile.readBitString(5)).bitPattern();
        if (numNonEmptyLevels == 0) {
            return;
        }

        // build leafsAtLevelList for getHuffmanCodes method
        int firstNonEmptyLevel =
(sourceFile.readBitString(5)).bitPattern();
        int lenLevel = (sourceFile.readBitString(5)).bitPattern();
        int numLevels = numNonEmptyLevels + firstNonEmptyLevel;
        BitString[] leafsAtLevelList = new BitString[numLevels];
        for (int i = 0; i < numLevels; i++) {
            if (i < firstNonEmptyLevel) {
                leafsAtLevelList[i] = new BitString((short)lenLevel,
0);
            } else {
                leafsAtLevelList[i] =
sourceFile.readBitString(lenLevel);
            }
```

```
        }

        // getHuffmanCodes
        BitString[] freqOrderedHuffmanCodes =
VonNoymanNode.getHuffmanCodes(leafsAtLevelList);

        // fill decodingMap with (Huffman code --> unencoded bit
string) mapping
        int numLeafs = freqOrderedHuffmanCodes.length;
        for (int i = 0; i < numLeafs; i++) {
            decodingMap.put(freqOrderedHuffmanCodes[i],
sourceFile.readBitString(n));
        }

        // find length of shortest Huffman code
        for (int i = 0; i < numLevels; i++) {
            if (leafsAtLevelList[i].bitPattern() != 0) {
                lenOfShortCodeInTable.value = i;
                return;
            }
        }
    }


}
```

```
package thesis.compression.multi_tree;


public class LengthOutOfBoundsException extends RuntimeException {
    public LengthOutOfBoundsException() {
        super("BitString length must be between 0 and " +
        Integer.toString(BitString.MAXLEN) + " (inclusive).");
    }

    public LengthOutOfBoundsException(String str) {
        super(str);
    }
}
```

```
package thesis.compression.lyndon;

import java.util.*;
import java.io.*;
import thesis.compression.multi_tree.*;

/**
```

```
 *  Preliminary investigation into the use of Lyndon words to build a
dictionary
 *  compression technique.  Not used in final thesis.  Looks promising
enough to continue
 *  researching.
 */
public class LyndonCompressionManager {

    ArrayList dictionary;
    int[] freq;
    BitStringReader sourceFile;
    BitStringWriter targetFile;
    int nBound;
    BitString remainder;
 // BitString offset;


    public static void main(String[] args) {

        String usage = new String("USAGE:  java
HuffmanCompressionManager " +
            "<sourceFilename> <targetFilename> <nBound>");

        try {
            if (args.length != 3) {
                throw new ArrayIndexOutOfBoundsException();
            }
            int nBound = Integer.parseInt(args[2]);
            BitStringReader sourceFile = new BitStringReader(args[0]);
            BitStringWriter targetFile = new BitStringWriter(args[1]);
            LyndonCompressionManager lcm =
                new LyndonCompressionManager(sourceFile, targetFile,
nBound);
            lcm.compress();
        } catch (NumberFormatException arg3NotInt) {
            System.out.println(usage);
        } catch (ArrayIndexOutOfBoundsException wrongNumArgs) {
            System.out.println(usage);
        } catch (LengthOutOfBoundsException paramsOutOfRange) {
            System.out.println("1 < nBound <= " + BitString.MAXLEN);
        } catch (Exception e) {
            e.printStackTrace(new PrintStream(System.out));
        } finally {
            System.exit(1);
        }
    }

    public LyndonCompressionManager(BitStringReader sourceFile,
                                    BitStringWriter targetFile,
                                    int nBound)
    throws
    IOException,
    LengthOutOfBoundsException
    {
        // bounds check
        if (nBound < 2 || nBound > BitString.MAXLEN) {
            throw new LengthOutOfBoundsException();
```

```
        }

        // initialize instance variables
        this.nBound = nBound;
        this.sourceFile = sourceFile;
        this.targetFile = targetFile;
        remainder = new BitString();
        dictionary = new ArrayList();
        freq = new int[100];
        Arrays.fill(freq, 1);
    }

    public void compress()
    throws
    IOException
    {
        loadDictionary(sourceFile, dictionary, nBound, remainder);
        // compressSourceFile(targetFile, sourceFile, freqTable, n,
remainder, offset);
    }

    public void loadDictionary(BitStringReader sourceFile,
                               ArrayList dictionary,
                               int nBound,
                               BitString remainder)
    throws
    IOException
    {
        int i;
        BitString lynWord1 = null;
        BitString lynWord2 = sourceFile.readLyndonWord(nBound);
        while (lynWord2.length() != 0) {
            i = dictionary.indexOf(lynWord1);
            if (i == -1) {
                dictionary.add(lynWord1);
            } else {
                freq[i]++;
            }
            lynWord1 = lynWord2;
            lynWord2 = sourceFile.readLyndonWord(nBound);
            i++;
        }
        remainder.reInit(lynWord1);
        printDictionary();
    }

    private void printDictionary() {
        int numBits = 0;
        int size = dictionary.size();
        int logSize = (int)Math.ceil(Math.log(size) / Math.log(2));
        BitString entry;
        for (int i = 1; i < size; i++) {
            entry = (BitString)dictionary.get(i);
            System.out.print(i + ".  " + (i < 10 ? " " : ""));
            System.out.print(entry);
            printSpaces(nBound - entry.length() + 3);
            System.out.println(freq[i]);
```

```
            numBits += (logSize * freq[i]);
        }
        System.out.println("remainder: " + remainder);
        System.out.println("source bits: " + sourceFile.bitsRead());
        System.out.println("target bits: " + numBits);
    }

    private void printSpaces(int num) {
        for (int i = 0; i < num; i++) {
            System.out.print(" ");
        }
    }
}
```

MULTITREE COMPRESSION FORMAT

```
n                                                                    5
bits

bits used Bf to represent # leafs f at each level of Huffman tree    5
bits
bits used Bv to represent # of levels v in Huffman tree              5
bits
leafs in level 0 of Huffman tree                                     Bf
bits
leafs in level 1 of Huffman tree                                     Bf
bits
.
.
.
leafs in level v - 1 of Huffman tree                                 Bf
bits
```

```
package thesis.compression.multi_tree;

import java.util.*;
import java.io.*;

/**
 *  This class implements the RTA approach presented in the thesis.
 */
public class MultitreeCompressionManager implements StatsCollected {

    protected Necklace neck;
    protected HashMap[] hashes;
    protected BitStringReader sourceFile;
    protected BitStringWriter targetFile;
    protected BitString remainder;
```

```
    protected BitString offset;
    protected boolean lastTreeInspectable;
    protected long headerSize;

    public static void main(String[] args) {

        String usage = new String("USAGE:  java
MulitreeCompressionManager " +
            "<sourceFilename> <targetFilename> <n> <offset>");

        try {
            if (args.length != 4) {
                throw new ArrayIndexOutOfBoundsException();
            }
            int n = Integer.parseInt(args[2]);
            int offset = Integer.parseInt(args[3]);
            BitStringReader sourceFile = new BitStringReader(args[0],
n);
            BitStringWriter targetFile = new BitStringWriter(args[1]);
            MultitreeCompressionManager mcm =
                new MultitreeCompressionManager(sourceFile, targetFile,
n, offset);
            mcm.compress();
        } catch (NumberFormatException arg3Or4NotInt) {
            System.out.println(usage);
        } catch (ArrayIndexOutOfBoundsException wrongNumArgs) {
            System.out.println(usage);
        } catch (LengthOutOfBoundsException paramsOutOfRange) {
            System.out.println("1 < n <= " + BitString.MAXLEN + "  AND
0 <= offset < n");
        } catch (Exception e) {
            e.printStackTrace(new PrintStream(System.out));
        } finally {
            System.exit(1);
        }
    }

    public MultitreeCompressionManager(BitStringReader sourceFile,
                                       BitStringWriter targetFile,
                                       int n,
                                       int offsetLen)
    throws
    IOException,
    LengthOutOfBoundsException
    {
        // bounds check
        if (n < 2 || n > BitString.MAXLEN || offsetLen < 0 || offsetLen
>= n) {
            throw new LengthOutOfBoundsException();
        }

        // initialize instance variables
        neck = new Necklace(n);
        this.sourceFile = sourceFile;
        this.targetFile = targetFile;
        sourceFile.setN(n);
        offset = sourceFile.readBitString(offsetLen);
```

```
        remainder = new BitString();
        lastTreeInspectable = false;
        headerSize = -1;
        hashes = new HashMap[n + 1];
        for (int i = 0; i <= n; i++) {
            hashes[i] = new HashMap();
        }
    }

    public void compress()
    throws
    IOException
    {
        buildFreqTables();
        compressSourceFile();
    }


    // ********** methods required to implement StatsCollected
interface **********

    public int iBits() {return neck.rotLength();}
    public int jBits() {return neck.indexLength();}
    public int iTree() {return neck.n();}
    public int n() {return neck.n();}
    public int lenOffset() {return offset.length();}
    public String sourceFilename() {return sourceFile.filename();}

    public long sourceFileSize()
    throws
    StatsNotAvailableException
    {
        long sfs = sourceFile.bitsRead();
        if (sfs > offset.length()) {
            return sfs;
        } else {
            throw new StatsNotAvailableException();
        }
    }

    public long targetFileSize()
    throws
    StatsNotAvailableException
    {
        long tfs = targetFile.bitsWrote();
        if (tfs > 0) {
            return tfs;
        } else {
            throw new StatsNotAvailableException();
        }
    }

    public Iterator lastTreeInspector()
    throws
    StatsNotAvailableException
    {
        if (lastTreeInspectable) {
```

```
            return hashes[neck.n()].values().iterator();
        } else {
            throw new StatsNotAvailableException();
        }
    }

    public long headerSize()
    throws
    StatsNotAvailableException
    {
        if (headerSize == -1) {
            throw new StatsNotAvailableException();
        } else {
            return headerSize;
        }
    }

    public int[] treeSizes()
    throws
    StatsNotAvailableException
    {
        int n = neck.n();
        if (hashes[n].isEmpty()) {
            throw new StatsNotAvailableException();
        } else {
            int[] numLeafs = new int[n];
            for (int i = 0; i < n; i++) {
                numLeafs[i] = hashes[i].isEmpty() ? 0 :
hashes[i].size();
            }
            return numLeafs;
        }
    }

    public void buildFreqTables()
    throws
    IOException
    {
        buildFreqTables(sourceFile, neck, hashes, remainder);
        lastTreeInspectable = true;
    }

    public void compressSourceFile()
    throws
    IOException
    {
        lastTreeInspectable = false;
        compressSourceFile(targetFile, sourceFile, neck, hashes,
remainder, offset);
    }

    // ************** end of methods for StatsCollected interface
****************


    protected void buildFreqTables( /* IN  */ BitStringReader
sourceFile,
```

```
                                    /* IN  */ Necklace neck,
                                    /* OUT */ HashMap[] freqTables,
                                    /* OUT */ BitString remainder)
    throws
    IOException
    {
        short indexLen = (short)neck.indexLength();
        short rotLen = (short)neck.rotLength();
        int n = neck.n();
        BitString bs = sourceFile.readBitString();

        // prescan entire source file and build n + 1 freqency tables
        while (bs.length() == n) {
            int bitPattern = bs.bitPattern();
            int classIndex = neck.bitStringToIndex(bitPattern);
            int rot = neck.bitStringToRots(bitPattern);

            // reuse BitString object to represent class index then
            // throw the class index into freqTable[0...n - 1]
            bs.reInit(indexLen, classIndex);
            if (freqTables[rot].containsKey(bs)) {
                ((VonNoymanNode)(freqTables[rot].get(bs))).incrFreq();
            } else {
                freqTables[rot].put(bs, new VonNoymanNode(bs));
            }

            // throw rotation into freqTable[n]
            BitString r = new BitString(rotLen, rot);
            if (freqTables[n].containsKey(r)) {
                ((VonNoymanNode)(freqTables[n].get(r))).incrFreq();
            } else {
                freqTables[n].put(r, new VonNoymanNode(r));
            }

            bs = sourceFile.readBitString();
        }
        remainder.reInit(bs);
    }

    protected void compressSourceFile(/* OUT    */ BitStringWriter
targetFile,
                                      /* IN     */ BitStringReader
sourceFile,
                                      /* IN     */ Necklace neck,
                                      /* IN/OUT */ HashMap[]
freqTables,
                                      /* IN     */ BitString remainder,
                                      /* IN     */ BitString offset)
    throws
    IOException
    {
        int n = neck.n();

        // write compression header
        writeCompressionHeader(targetFile, sourceFile, n, freqTables,
remainder, offset);
```

```
        HashMap[] encodingMap = freqTables;
        sourceFile.reset();
        sourceFile.readBitString(offset.length());  // throw away
offset

        // write sourceFile to targetFile in compressed form
        int indexLen = neck.indexLength();
        int rotLen = neck.rotLength();
        BitString bs1 = sourceFile.readBitString();
        BitString bs2 = new BitString();

        while (bs1.length() == n) {

            int bitPattern = bs1.bitPattern();
            int classIndex = neck.bitStringToIndex(bitPattern);
            int rot = neck.bitStringToRots(bitPattern);

            // write Huffman code for rotations to targetFile
            // recycling BitString objects to avoid unnecessary object
creation
            bs1.reInit(rotLen, rot);
            bs2.reInit((BitString)encodingMap[n].get(bs1));
            targetFile.writeBitString(bs2);

            // write Huffman code for class index to targetFile
            // recycling BitString objects
            bs1.reInit(indexLen, classIndex);
            bs2.reInit((BitString)encodingMap[rot].get(bs1));
            targetFile.writeBitString(bs2);

            bs1 = sourceFile.readBitString();
        }
        Assertion.assert(remainder.equals(bs1));
        targetFile.close();
    }

    protected void writeCompressionHeader(/* OUT    */ BitStringWriter
targetFile,
                                          /* IN     */ BitStringReader
sourceFile,
                                          /* IN     */ int n,
                                          /* IN/OUT */ HashMap[]
freqTables,
                                          /* IN     */ BitString
remainder,
                                          /* IN     */ BitString
offset)
    throws
    IOException
    {
        // first entries in header of target file
        targetFile.writeBitString(new BitString(5, n));
        targetFile.writeBitString(new BitString(5, offset.length()));
        targetFile.writeBitString(offset);
        targetFile.writeBitString(new BitString(5,
remainder.length()));
        targetFile.writeBitString(remainder);
```

```
        // convert each HashMap from a freqency table to an encoding
map and write
        // header info for each Huffman tree
        for (int i = 0; i <= n; i++) {
            processTree(freqTables[i], targetFile);
        }

        // write numBitStringsRead as last entry in header
        long numBitStringsRead = (sourceFile.bitsRead() -
offset.length()) / n;
        Assertion.assert(numBitStringsRead <= Integer.MAX_VALUE);
        targetFile.writeBitString(new BitString(32,
(int)numBitStringsRead));
        headerSize = targetFile.bitsWrote();
    }

    protected void processTree(/*  IN/OUT  */  HashMap freqTable,
                               /*  OUT     */  BitStringWriter
targetFile)
    throws
    IOException
    {
        // if the freq table is empty write minimal header info and
return
        if (freqTable.isEmpty()) {
            targetFile.writeBitString(new BitString(10, 0));
            return;
        }

        // order all the VonNoymanNodes of this table based on
frequency
        TreeSet freqOrderedVNN = new TreeSet(freqTable.values());

        // build a frequency ordered class index list for the
compression header
        BitString[] freqOrderedClassIndexList = new
BitString[freqOrderedVNN.size()];
        Iterator it = freqOrderedVNN.iterator();
        int i = 0;
        while (it.hasNext()) {
            freqOrderedClassIndexList[i++] =
((VonNoymanNode)it.next()).index;
        }

        // build leafs at level list for the compression header
        // NOTE:  this algorithm trashes its TreeSet parameter AND the
underlying
        // HashMap upon which it is based)
        BitString[] leafsAtLevelList =
VonNoymanNode.vonNoymanAlgorithm(freqOrderedVNN);

        // write this tree's portion of the compression header
        BitString leafsPerLevel = new BitString(5,
leafsAtLevelList[0].length());
        BitString  numLevels = new BitString(5,
leafsAtLevelList.length);
```

```
        targetFile.writeBitString(leafsPerLevel);
        targetFile.writeBitString(numLevels);
        targetFile.writeBitString(leafsAtLevelList);
        targetFile.writeBitString(freqOrderedClassIndexList);

        // build frequency ordered list of Huffman codes
        BitString[] freqOrderedHuffmanCodes =
VonNoymanNode.getHuffmanCodes(leafsAtLevelList);

        // reuse the trashed HashMap as a (class index --> Huffman
code) encoding map
        HashMap classIndexToHuffmanCodeMap = freqTable;
        for (int j = 0; j < freqOrderedClassIndexList.length; j++) {
            classIndexToHuffmanCodeMap.put(
                freqOrderedClassIndexList[j],
freqOrderedHuffmanCodes[j]);
        }
    }
}



package thesis.compression.multi_tree2;

import java.util.*;
import java.io.*;
import thesis.compression.multi_tree.*;

/**
 *  This class implements the ITA approach discussed in the thesis.
 */
public class MultitreeCompressionManager2 implements StatsCollected {

    protected HashMap[] hashes;
    protected BitStringReader sourceFile;
    protected BitStringWriter targetFile;
    protected int iBits;
    protected int jBits;
    protected BitString remainder;
    protected BitString offset;
    protected int iTree;
    protected boolean lastTreeInspectable;
    protected long headerSize;

    public static void main(String[] args) {

        String usage = "USAGE:  java MulitreeCompressionManager2 " +
        "<sourceFilename> <targetFilename> <iBits> <jBits> <offset>";

        try {
            if (args.length != 5) {
                throw new ArrayIndexOutOfBoundsException();
            }
            int iBits = Integer.parseInt(args[2]);
            int jBits = Integer.parseInt(args[3]);
            int offset = Integer.parseInt(args[4]);
```

```
            BitStringReader sourceFile = new BitStringReader(args[0]);
            BitStringWriter targetFile = new BitStringWriter(args[1]);
            MultitreeCompressionManager2 mcm2 = new
MultitreeCompressionManager2(
                sourceFile, targetFile, iBits, jBits, offset);
            mcm2.compress();
        } catch (NumberFormatException arg2or3or4NotInt) {
            System.out.println(usage);
        } catch (ArrayIndexOutOfBoundsException wrongNumArgs) {
            System.out.println(usage);
        } catch (LengthOutOfBoundsException paramsOutOfRange) {
            System.out.println("1 < iBits + jBits <= " +
            BitString.MAXLEN + "  AND  0 <= offset < iBits + jBits");
        } catch (Exception e) {
            e.printStackTrace(new PrintStream(System.out));
        } finally {
            System.exit(1);
        }
    }

    public MultitreeCompressionManager2(BitStringReader sourceFile,
                                        BitStringWriter targetFile,
                                        int iBits,
                                        int jBits,
                                        int offsetLen)
    throws
    IOException,
    LengthOutOfBoundsException
    {
        // bounds check
        if (iBits < 1 || jBits < 1 || iBits + jBits > BitString.MAXLEN
||
            offsetLen < 0 || offsetLen >= iBits + jBits) {
            throw new LengthOutOfBoundsException();
        }

        // initialize instance variables
        this.sourceFile = sourceFile;
        this.targetFile = targetFile;
        this.iBits = iBits;
        this.jBits = jBits;
        offset = sourceFile.readBitString(offsetLen);
        remainder = new BitString();
        iTree = (int)(Math.pow(2, iBits));
        lastTreeInspectable = false;
        headerSize = -1;
        hashes = new HashMap[iTree + 1];
        for (int i = 0; i <= iTree; i++) {
            hashes[i] = new HashMap();
        }
    }

    // *********** methods required to implement StatsCollected
interface ***********

    public int iBits() {return iBits;}
    public int jBits() {return jBits;}
```

```
public int n() {return iBits + jBits;}
public int iTree() {return iTree;}
public int lenOffset() {return offset.length();}
public String sourceFilename() {return sourceFile.filename();}

public long sourceFileSize()
throws
StatsNotAvailableException
{
    long sfs = sourceFile.bitsRead();
    if (sfs > offset.length()) {
        return sfs;
    } else {
        throw new StatsNotAvailableException();
    }
}

public long headerSize()
throws
StatsNotAvailableException
{
    if (headerSize == -1) {
        throw new StatsNotAvailableException();
    } else {
        return headerSize;
    }
}


public long targetFileSize()
throws
StatsNotAvailableException
{
    long tfs = targetFile.bitsWrote();
    if (tfs > 0) {
        return tfs;
    } else {
        throw new StatsNotAvailableException();
    }
}

public Iterator lastTreeInspector()
throws
StatsNotAvailableException
{
    if (lastTreeInspectable) {
        return hashes[iTree].values().iterator();
    } else {
        throw new StatsNotAvailableException();
    }
}

public int[] treeSizes()
throws
StatsNotAvailableException
{
    if (hashes[iTree].isEmpty()) {
```

```
            throw new StatsNotAvailableException();
        } else {
            int[] numLeafs = new int[iTree];
            for (int i = 0; i < iTree; i++) {
                numLeafs[i] = hashes[i].isEmpty() ? 0 :
hashes[i].size();
            }
            return numLeafs;
        }
    }

    public void buildFreqTables() throws IOException {
        buildFreqTables(sourceFile, hashes, iBits, jBits, iTree,
remainder);
        lastTreeInspectable = true;
    }

    public void compressSourceFile() throws IOException {
        lastTreeInspectable = false;
        compressSourceFile(targetFile, sourceFile, hashes, iBits,
jBits, iTree, remainder, offset);
    }

    // ************** end of methods for StatsCollected interface
***************

    public void compress()
    throws
    IOException
    {
        buildFreqTables();
        compressSourceFile();
    }

    protected void buildFreqTables( /* IN  */ BitStringReader
sourceFile,
                                    /* OUT */ HashMap[] freqTables,
                                    /* IN  */ int iBits,
                                    /* IN  */ int jBits,
                                    /* IN  */ int iTree,
                                    /* OUT */ BitString remainder)
    throws
    IOException
    {
        int lenWord = iBits + jBits;
        BitString peekString = sourceFile.peekBitString(lenWord);

        // prescan entire source file and build iTree + 1 freqency
tables
        while (peekString.length() == lenWord) {

            BitString iWord = sourceFile.readBitString(iBits);
            BitString jWord = sourceFile.readBitString(jBits);

            if (freqTables[iWord.bitPattern()].containsKey(jWord)) {
```

```
((VonNoymanNode)(freqTables[iWord.bitPattern()].get(jWord))).incrFreq()
;
            } else {
                freqTables[iWord.bitPattern()].put(jWord, new
VonNoymanNode(jWord));
            }

            // throw iWord into freqTable[iTree]
            if (freqTables[iTree].containsKey(iWord)) {

((VonNoymanNode)(freqTables[iTree].get(iWord))).incrFreq();
            } else {
                freqTables[iTree].put(iWord, new VonNoymanNode(iWord));
            }

            peekString = sourceFile.peekBitString(lenWord);
        }
        remainder.reInit(peekString);
        sourceFile.readBitString(peekString.length());  // read and
throw away remainder
                                                        // to keep
bitsRead() accurate
    }

    protected void compressSourceFile(/* OUT    */ BitStringWriter
targetFile,
                                      /* IN     */ BitStringReader
sourceFile,
                                      /* IN/OUT */ HashMap[]
freqTables,
                                      /* IN     */ int iBits,
                                      /* IN     */ int jBits,
                                      /* IN     */ int iTree,
                                      /* IN     */ BitString remainder,
                                      /* IN     */ BitString offset)
    throws
    IOException
    {
        writeCompressionHeader(targetFile, sourceFile, iBits, jBits,
iTree, freqTables, remainder, offset);

        HashMap[] encodingMap = freqTables;          // reusing data
structure
        sourceFile.reset();                          // read the file
AGAIN
        sourceFile.readBitString(offset.length());  // throw away
offset

        int lenWord = iBits + jBits;
        BitString peekString = sourceFile.peekBitString(lenWord);
        while (peekString.length() == lenWord) {

            BitString iWord = sourceFile.readBitString(iBits);
            BitString jWord = sourceFile.readBitString(jBits);

            // write Huffman code for iWord to targetFile
```

```
targetFile.writeBitString((BitString)encodingMap[iTree].get(iWord));

            // write Huffman code for jword to targetFile

targetFile.writeBitString((BitString)encodingMap[iWord.bitPattern()].ge
t(jWord));

            peekString = sourceFile.peekBitString(lenWord);
        }

        Assertion.assert(remainder.equals(peekString));
        targetFile.close();
    }

    protected void writeCompressionHeader(/* OUT    */ BitStringWriter
targetFile,
                                          /* IN     */ BitStringReader
sourceFile,
                                          /* IN     */ int iBits,
                                          /* IN     */ int jBits,
                                          /* IN     */ int iTree,
                                          /* IN/OUT */ HashMap[]
freqTables,
                                          /* IN     */ BitString
remainder,
                                          /* IN     */ BitString
offset)
    throws
    IOException
    {
        // first entries in header of target file
        targetFile.writeBitString(new BitString((short)5, iBits));
        targetFile.writeBitString(new BitString((short)5, jBits));
        targetFile.writeBitString(new BitString((short)5,
offset.length()));
        targetFile.writeBitString(offset);
        targetFile.writeBitString(new BitString((short)5,
remainder.length()));
        targetFile.writeBitString(remainder);

        // convert each HashMap from a freqency table to an encoding
map and write
        // header info for each Huffman tree
        for (int i = 0; i <= iTree; i++) {
            processTree(freqTables[i], targetFile);
        }

        // write numBitStringsRead as last entry in header
        long numBitStringsRead = (sourceFile.bitsRead() -
offset.length()) / (iBits + jBits);
        Assertion.assert(numBitStringsRead <= Integer.MAX_VALUE);
        targetFile.writeBitString(new BitString((short)32,
(int)numBitStringsRead));
        headerSize = targetFile.bitsWrote();
    }
```

```
    protected void processTree(/*  IN/OUT  */  HashMap freqTable,
                               /*  OUT     */  BitStringWriter
targetFile)
    throws
    IOException
    {
        // if the freq table is empty write minimal header info and
return
        if (freqTable.isEmpty()) {
            targetFile.writeBitString(new BitString((short)10, 0));
            return;
        }

        // order all the VonNoymanNodes of this table based on
frequency
        TreeSet freqOrderedVNN = new TreeSet(freqTable.values());

        // build a frequency ordered j-word list for the compression
header
        BitString[] freqOrderedJWordList = new
BitString[freqOrderedVNN.size()];
        Iterator it = freqOrderedVNN.iterator();
        int i = 0;
        while (it.hasNext()) {
            freqOrderedJWordList[i++] =
((VonNoymanNode)it.next()).index;
        }

        // build leafs at level list for the compression header
        // NOTE:  this algorithm trashes its TreeSet parameter AND the
underlying
        // HashMap upon which it is based)
        BitString[] leafsAtLevelList =
VonNoymanNode.vonNoymanAlgorithm(freqOrderedVNN);

        // write this tree's portion of the compression header
        BitString leafsPerLevel = new BitString((short)5,
leafsAtLevelList[0].length());
        BitString  numLevels = new BitString((short)5,
leafsAtLevelList.length);
        targetFile.writeBitString(leafsPerLevel);
        targetFile.writeBitString(numLevels);
        targetFile.writeBitString(leafsAtLevelList);
        targetFile.writeBitString(freqOrderedJWordList);

        // build frequency ordered list of Huffman codes
        BitString[] freqOrderedHuffmanCodes =
VonNoymanNode.getHuffmanCodes(leafsAtLevelList);

        // reuse the trashed HashMap as a (j-word --> Huffman code)
encoding map
        HashMap JWordToHuffmanCodeMap = freqTable;
        for (int k = 0; k < freqOrderedJWordList.length; k++) {
            JWordToHuffmanCodeMap.put(freqOrderedJWordList[k],
freqOrderedHuffmanCodes[k]);
        }
    }
```

```
}
```

```
package thesis.compression.multi_tree3;

import java.util.*;
import java.io.*;
import thesis.compression.multi_tree.*;
import thesis.compression.multi_tree2.*;

/**
 *  This class is identical to MultitreeCompressionManager2 (ITA)
except that it uses
 *  a slightly more efficient header encoding format.  The performance
difference between the two
 *  techniques is typically insignificant (<0.1%).
 */
public class MultitreeCompressionManager3
extends MultitreeCompressionManager2
implements StatsCollected
{
    public static void main(String[] args) {

        String usage = "USAGE:  java MulitreeCompressionManager3 " +
        "<sourceFilename> <targetFilename> <iBits> <jBits> <offset>";

        try {
            if  (args.length != 5) {
                throw new ArrayIndexOutOfBoundsException();
            }
            int iBits = Integer.parseInt(args[2]);
            int jBits = Integer.parseInt(args[3]);
            int offset = Integer.parseInt(args[4]);
            BitStringReader sourceFile = new BitStringReader(args[0]);
            BitStringWriter targetFile = new BitStringWriter(args[1]);
            MultitreeCompressionManager3 mcm3 = new
MultitreeCompressionManager3(
                sourceFile, targetFile, iBits, jBits, offset);
            mcm3.compress();
        } catch (NumberFormatException arg2or3or4NotInt) {
            System.out.println(usage);
        } catch (ArrayIndexOutOfBoundsException wrongNumArgs) {
            System.out.println(usage);
        } catch (LengthOutOfBoundsException paramsOutOfRange) {
            System.out.println("1 < iBits + jBits <= " +
            BitString.MAXLEN + "  AND  0 <= offset < iBits + jBits");
        } catch (Exception e) {
            e.printStackTrace(new PrintStream(System.out));
        } finally {
            System.exit(1);
        }
    }

    public MultitreeCompressionManager3(BitStringReader sourceFile,
```

```
                                        BitStringWriter targetFile,
                                        int iBits,
                                        int jBits,
                                        int offsetLen)
    throws
    IOException,
    LengthOutOfBoundsException
    {
        super(sourceFile, targetFile, iBits, jBits, offsetLen);
    }

    protected void processTree(/*  IN/OUT  */  HashMap freqTable,
                               /*  OUT     */  BitStringWriter
targetFile)
    throws
    IOException
    {
        // if the freq table is empty write minimal header info and
return
        if (freqTable.isEmpty()) {
            targetFile.writeBitString(new BitString((short)5, 0));
            return;
        }

        // order all the VonNoymanNodes of this table based on
frequency
        TreeSet freqOrderedVNN = new TreeSet(freqTable.values());

        // build a frequency ordered j-word list for the compression
header
        BitString[] freqOrderedJWordList = new
BitString[freqOrderedVNN.size()];
        Iterator it = freqOrderedVNN.iterator();
        int i = 0;
        while (it.hasNext()) {
            freqOrderedJWordList[i++] =
((VonNoymanNode)it.next()).index;
        }

        // build and trim leafs at level list for the compression
header
        // NOTE:  the VonNoyman algorithm trashes its TreeSet parameter
AND
        // the underlying HashMap upon which it is based)
        BitString[] leafsAtLevelList =
VonNoymanNode.vonNoymanAlgorithm(freqOrderedVNN);
        BitString[] trimedLeafsAtLevelList =
trimEmptyLeadLevels(leafsAtLevelList);
        int firstLevel = leafsAtLevelList.length -
trimedLeafsAtLevelList.length;

        // write this tree's portion of the compression header
        BitString numNonEmptyLevels = new BitString((short)5,
trimedLeafsAtLevelList.length);
        BitString firstNonEmptyLevel = new BitString((short)5,
firstLevel);
```

```
        BitString leafsPerLevel = new BitString((short)5,
trimedLeafsAtLevelList[0].length());
        targetFile.writeBitString(numNonEmptyLevels);
        targetFile.writeBitString(firstNonEmptyLevel);
        targetFile.writeBitString(leafsPerLevel);
        targetFile.writeBitString(trimedLeafsAtLevelList);
        targetFile.writeBitString(freqOrderedJWordList);

        // build frequency ordered list of Huffman codes
        BitString[] freqOrderedHuffmanCodes =
VonNoymanNode.getHuffmanCodes(leafsAtLevelList);

        // reuse the trashed HashMap as a (j-word --> Huffman code)
encoding map
        HashMap JWordToHuffmanCodeMap = freqTable;
        for (int k = 0; k < freqOrderedJWordList.length; k++) {
            JWordToHuffmanCodeMap.put(freqOrderedJWordList[k],
freqOrderedHuffmanCodes[k]);
        }
    }

    private final BitString[] trimEmptyLeadLevels(BitString[]
leafsAtLevelList) {
        int numNonEmptyLevels = leafsAtLevelList.length;
        int i = 0;
        while (leafsAtLevelList[i].bitPattern() == 0) {
            numNonEmptyLevels--;
            i++;
        }

        BitString[] newLeafsAtLevelList = new
BitString[numNonEmptyLevels];
        System.arraycopy(leafsAtLevelList, i, newLeafsAtLevelList, 0,
numNonEmptyLevels);
        return newLeafsAtLevelList;
    }
}
```

```
package thesis.compression.multi_tree4;

import java.util.*;
import java.io.*;
import thesis.compression.multi_tree.*;
import thesis.compression.multi_tree3.*;

/**
 *  This class is identical to MultitreeCompressionManager3 except that
it uses fixed
 *  length iBit codes instead of Huffman codes from an iBit tree.  It
was an experiment
 *  to see how much of an effect the iBit tree was having on our
overall compression
 *  results.  This code helped us build our theoretical model of ITA.
```

```
 */
public class MultitreeCompressionManager4
extends MultitreeCompressionManager3
implements StatsCollected
{
    public static void main(String[] args) {

        String usage = "USAGE:  java MulitreeCompressionManager4 " +
        "<sourceFilename> <targetFilename> <iBits> <jBits> <offset>";

        try {
            if (args.length != 5) {
                throw new ArrayIndexOutOfBoundsException();
            }
            int iBits = Integer.parseInt(args[2]);
            int jBits = Integer.parseInt(args[3]);
            int offset = Integer.parseInt(args[4]);
            BitStringReader sourceFile = new BitStringReader(args[0]);
            BitStringWriter targetFile = new BitStringWriter(args[1]);
            MultitreeCompressionManager4 mcm4 = new
MultitreeCompressionManager4(
                sourceFile, targetFile, iBits, jBits, offset);
            mcm4.compress();
        } catch (NumberFormatException arg2or3or4NotInt) {
            System.out.println(usage);
        } catch (ArrayIndexOutOfBoundsException wrongNumArgs) {
            System.out.println(usage);
        } catch (LengthOutOfBoundsException paramsOutOfRange) {
            System.out.println("1 < iBits + jBits <= " +
            BitString.MAXLEN + "  AND  0 <= offset < iBits + jBits");
        } catch (Exception e) {
            e.printStackTrace(new PrintStream(System.out));
        } finally {
            System.exit(1);
        }
    }

    public MultitreeCompressionManager4(BitStringReader sourceFile,
                                        BitStringWriter targetFile,
                                        int iBits,
                                        int jBits,
                                        int offsetLen)
    throws
    IOException,
    LengthOutOfBoundsException
    {
        super(sourceFile, targetFile, iBits, jBits, offsetLen);
    }

    protected void compressSourceFile(/* OUT    */ BitStringWriter
targetFile,
                                      /* IN     */ BitStringReader
sourceFile,
                                      /* IN/OUT */ HashMap[]
freqTables,
                                      /* IN     */ int iBits,
                                      /* IN     */ int jBits,
```

```
                                        /* IN     */ int iTree,
                                        /* IN     */ BitString remainder,
                                        /* IN     */ BitString offset)
    throws
    IOException
    {
        writeCompressionHeader(targetFile, sourceFile, iBits, jBits,
iTree, freqTables, remainder, offset);

        HashMap[] encodingMap = freqTables;         // reusing data
structure
        sourceFile.reset();                         // read the file
AGAIN
        sourceFile.readBitString(offset.length()); // throw away
offset

        int lenWord = iBits + jBits;
        BitString peekString = sourceFile.peekBitString(lenWord);
        while (peekString.length() == lenWord) {

            BitString iWord = sourceFile.readBitString(iBits);
            BitString jWord = sourceFile.readBitString(jBits);

            // write iWord (unencoded) to targetFile
            targetFile.writeBitString(iWord);

            // write Huffman code for jword to targetFile

targetFile.writeBitString((BitString)encodingMap[iWord.bitPattern()].ge
t(jWord));

            peekString = sourceFile.peekBitString(lenWord);
        }

        Assertion.assert(remainder.equals(peekString));
        targetFile.close();
    }

    protected void writeCompressionHeader(/* OUT    */ BitStringWriter
targetFile,
                                          /* IN     */ BitStringReader
sourceFile,
                                          /* IN     */ int iBits,
                                          /* IN     */ int jBits,
                                          /* IN     */ int iTree,
                                          /* IN/OUT */ HashMap[]
freqTables,
                                          /* IN     */ BitString
remainder,
                                          /* IN     */ BitString
offset)
    throws
    IOException
    {
        // first entries in header of target file
        targetFile.writeBitString(new BitString((short)5, iBits));
        targetFile.writeBitString(new BitString((short)5, jBits));
```

```
        targetFile.writeBitString(new BitString((short)5,
offset.length()));
        targetFile.writeBitString(offset);
        targetFile.writeBitString(new BitString((short)5,
remainder.length()));
        targetFile.writeBitString(remainder);

        // convert each HashMap from a freqency table to an encoding
map and write
        // header info for each Huffman tree (EXCLUDING the iTree)
        for (int i = 0; i < iTree; i++) {
            processTree(freqTables[i], targetFile);
        }

        // write numWordsRead as last entry in header
        long numWordsRead = (sourceFile.bitsRead() - offset.length()) /
(iBits + jBits);
        Assertion.assert(numWordsRead <= Integer.MAX_VALUE);
        targetFile.writeBitString(new BitString((short)32,
(int)numWordsRead));
        headerSize = targetFile.bitsWrote();
    }
}



package thesis.compression.multi_tree;


import java.util.*;
import java.io.*;
import org.omg.CORBA.IntHolder;

/**
 *  This class decompresses files compressed by the RTA approach
presented in the thesis.
 */
public class MultitreeDecompressionManager {

    private BitStringWriter targetFile;
    private BitStringReader sourceFile;
    private HashMap[] decodingMaps;
    private int n;
    private BitString remainder;
    private BitString offset;


    public static void main(String[] args) {

        String usage = new String("USAGE:  java
MulitreeDecompressionManager " +
            "<sourceFilename> <targetFilename>");

        try {
            if (args.length != 2) {
                throw new ArrayIndexOutOfBoundsException();
```

```
            }
            BitStringReader sourceFile = new BitStringReader(args[0]);
            BitStringWriter targetFile = new BitStringWriter(args[1]);
            new MultitreeDecompressionManager(sourceFile, targetFile);
        } catch (ArrayIndexOutOfBoundsException wrongNumArgs) {
            System.out.println(usage);
        } catch (Exception e) {
            e.printStackTrace(new PrintStream(System.out));
        } finally {
            System.exit(1);
        }
    }

    public MultitreeDecompressionManager(BitStringReader sourceFile,
                                         BitStringWriter targetFile)
    throws
    IOException
    {
        this.sourceFile = sourceFile;
        this.targetFile = targetFile;
        n = (sourceFile.readBitString(5)).bitPattern();
        int lenOffset = (sourceFile.readBitString(5)).bitPattern();
        offset = sourceFile.readBitString(lenOffset);
        int lenRem = (sourceFile.readBitString(5)).bitPattern();
        remainder = sourceFile.readBitString(lenRem);
        decodingMaps = new HashMap[n + 1];
        for (int i = 0; i <= n; i++) {
            decodingMaps[i] = new HashMap();
        }
        decompressSourceFile(targetFile, sourceFile, n, decodingMaps,
remainder, offset);
    }

    protected void decompressSourceFile(/* OUT */  BitStringWriter
targetFile,
                                        /* IN  */  BitStringReader
sourceFile,
                                        /* IN  */  int n,
                                        /* OUT */  HashMap[]
decodingMaps,
                                        /* IN  */  BitString remainder,
                                        /* IN  */  BitString offset)
    throws
    IOException
    {
        targetFile.writeBitString(offset);

        // build decoding maps (Huffman code ==> class index)
        // remember length of shortest Huffman code in each map
        IntHolder[] lenOfShortHufCode = new IntHolder[n + 1];
        for (int i = 0; i <= n; i++) {
            lenOfShortHufCode[i] = new IntHolder();
            buildDecodingMap(sourceFile, n, decodingMaps[i],
lenOfShortHufCode[i], i == n);
        }
```

```
        // build decoding table (necklace class index ==> necklace
class)
        long numClasses = Necklace.numClasses(n);
        Assertion.assert(numClasses <= Integer.MAX_VALUE);
        int[] indexToClass = new int[(int)numClasses];
        Necklace.loadTable_indexToClass(n, indexToClass);

        long numBitStringsCompressed =
(sourceFile.readBitString(32)).bitPattern();
        sourceFile.setDecodeLimit(2L * numBitStringsCompressed);
        sourceFile.setN(n);
        BitString r = null, c = null;
        int rot = -1;

        // get first encoded rotation in sourceFile
        r = sourceFile.decodeBitString(decodingMaps[n],
lenOfShortHufCode[n].value);

        // read an encoded rotation and class index from sourceFile
        // write an unencoded bitstring to targetFile
        while (r.length() > 0) {

            rot = r.bitPattern();

            // c is class index
            c = sourceFile.decodeBitString(decodingMaps[rot],
lenOfShortHufCode[rot].value);

            // c is class
            c.reInit(n, indexToClass[c.bitPattern()]);

            // c is original unencoded bitstring !!!
            c.rShift(rot, true);

            // write c to the target file
            targetFile.writeBitString(c);

            // get next encoded rotation in sourceFile
            // a BitString of length 0 indicates end of sourceFile
            r = sourceFile.decodeBitString(decodingMaps[n],
lenOfShortHufCode[n].value);
        }
        sourceFile.close();
        targetFile.writeBitString(remainder);
        targetFile.close();
    }


    protected void buildDecodingMap(/* IN  */ BitStringReader
sourceFile,
                                    /* IN  */ int n,
                                    /* OUT */ HashMap decodingMap,
                                    /* OUT */ IntHolder
lenOfShortCodeInTable,
                                    /* IN  */ boolean isRotMap)
    throws
    IOException
```

```
    {
        // build leafsAtLevelList for getHuffmanCodes method
        int lenLevel = (sourceFile.readBitString(5)).bitPattern();
        int numLevels = (sourceFile.readBitString(5)).bitPattern();
        BitString[] leafsAtLevelList = new BitString[numLevels];
        for (int i = 0; i < numLevels; i++) {
            leafsAtLevelList[i] = sourceFile.readBitString(lenLevel);
        }

        // getHuffmanCodes
        BitString[] freqOrderedHuffmanCodes =
            VonNoymanNode.getHuffmanCodes(leafsAtLevelList);

        // fill decodingMap with (Huffman code --> class index) mapping
        int lenIndex = isRotMap ? Necklace.rotLength(n) :
Necklace.indexLength(n);
        int numLeafs = freqOrderedHuffmanCodes.length;
        for (int i = 0; i < numLeafs; i++) {
            decodingMap.put(freqOrderedHuffmanCodes[i],
sourceFile.readBitString(lenIndex));
        }

        // find length of shortest Huffman code
        for (int i = 0; i < numLevels; i++) {
            if (leafsAtLevelList[i].bitPattern() != 0) {
                lenOfShortCodeInTable.value = i;
                return;
            }
        }
    }
}



package thesis.compression.multi_tree2;


import java.util.*;
import java.io.*;
import org.omg.CORBA.IntHolder;
import thesis.compression.multi_tree.*;

/**
 *  An instance of this class decompresses files compressed by
MultitreeCompressionManager2
 */
public class MultitreeDecompressionManager2 {

    private BitStringWriter targetFile;
    private BitStringReader sourceFile;
    private HashMap[] decodingMaps;
    private int iBits;
    private int jBits;
    private int iTree;
    private BitString remainder;
    private BitString offset;
```

```
    public static void main(String[] args) {

        String usage = new String("USAGE:  java
MulitreeDecompressionManager2 " +
            "<sourceFilename> <targetFilename>");

        try {
            if (args.length != 2) {
                throw new ArrayIndexOutOfBoundsException();
            }
            BitStringReader sourceFile = new BitStringReader(args[0]);
            BitStringWriter targetFile = new BitStringWriter(args[1]);
            new MultitreeDecompressionManager2(sourceFile, targetFile);
        } catch (ArrayIndexOutOfBoundsException wrongNumArgs) {
            System.out.println(usage);
        } catch (Exception e) {
            e.printStackTrace(new PrintStream(System.out));
        } finally {
            System.exit(1);
        }
    }

    public MultitreeDecompressionManager2(BitStringReader sourceFile,
                                          BitStringWriter targetFile)
    throws
    IOException
    {
        this.sourceFile = sourceFile;
        this.targetFile = targetFile;
        iBits = (sourceFile.readBitString(5)).bitPattern();
        jBits = (sourceFile.readBitString(5)).bitPattern();
        int lenOffset = (sourceFile.readBitString(5)).bitPattern();
        offset = sourceFile.readBitString(lenOffset);
        int lenRem = (sourceFile.readBitString(5)).bitPattern();
        remainder = sourceFile.readBitString(lenRem);
        iTree = (int)(Math.pow(2, iBits));
        decodingMaps = new HashMap[iTree + 1];
        for (int i = 0; i <= iTree; i++) {
            decodingMaps[i] = new HashMap();
        }
        decompressSourceFile(targetFile, sourceFile, iBits, jBits,
iTree, decodingMaps, remainder, offset);
    }

    protected void decompressSourceFile(/* OUT */  BitStringWriter
targetFile,
                                        /* IN  */  BitStringReader
sourceFile,
                                        /* IN  */  int iBits,
                                        /* IN  */  int jBits,
                                        /* IN  */  int iTree,
                                        /* OUT */  HashMap[]
decodingMaps,
                                        /* IN  */  BitString remainder,
                                        /* IN  */  BitString offset)
```

```
    throws
    IOException
    {
        targetFile.writeBitString(offset);

        // build decoding maps (Huffman code ==> jWord)
        // remember length of shortest Huffman code in each map
        IntHolder[] lenOfShortHufCode = new IntHolder[iTree + 1];
        for (int i = 0; i <= iTree; i++) {
            lenOfShortHufCode[i] = new IntHolder();
            buildDecodingMap(sourceFile, iBits, jBits, decodingMaps[i],
lenOfShortHufCode[i], i == iTree);
        }

        long numBitStringsCompressed =
(sourceFile.readBitString(32)).bitPattern();
        sourceFile.setDecodeLimit(2L * numBitStringsCompressed);
        BitString iWord = null, jWord = null;


        // get first encoded iWord in sourceFile
        iWord = sourceFile.decodeBitString(decodingMaps[iTree],
lenOfShortHufCode[iTree].value);

        // read an encoded iWord and jWord from sourceFile
        // write a decoded iWord and jWord to targetFile
        while (iWord.length() > 0) {

            jWord =
sourceFile.decodeBitString(decodingMaps[iWord.bitPattern()],
lenOfShortHufCode[iWord.bitPattern()].value);
            targetFile.writeBitString(iWord);
            targetFile.writeBitString(jWord);
            iWord = sourceFile.decodeBitString(decodingMaps[iTree],
lenOfShortHufCode[iTree].value);
        }

        sourceFile.close();
        targetFile.writeBitString(remainder);
        targetFile.close();
    }


    protected void buildDecodingMap(/* IN  */ BitStringReader
sourceFile,
                                    /* IN  */ int iBits,
                                    /* IN  */ int jBits,
                                    /* OUT */ HashMap decodingMap,
                                    /* OUT */ IntHolder
lenOfShortCodeInTable,
                                    /* IN  */ boolean isIWordMap)
    throws
    IOException
    {
        // build leafsAtLevelList for getHuffmanCodes method
        int lenLevel = (sourceFile.readBitString(5)).bitPattern();
        int numLevels = (sourceFile.readBitString(5)).bitPattern();
```

```
        BitString[] leafsAtLevelList = new BitString[numLevels];
        for (int i = 0; i < numLevels; i++) {
            leafsAtLevelList[i] = sourceFile.readBitString(lenLevel);
        }

        // getHuffmanCodes
        BitString[] freqOrderedHuffmanCodes =
VonNoymanNode.getHuffmanCodes(leafsAtLevelList);

        // fill decodingMap with (Huffman code --> jWord) mapping
        int bits2Read = isIWordMap ? iBits : jBits;
        int numLeafs = freqOrderedHuffmanCodes.length;
        for (int i = 0; i < numLeafs; i++) {
            decodingMap.put(freqOrderedHuffmanCodes[i],
sourceFile.readBitString(bits2Read));
        }

        // find length of shortest Huffman code
        for (int i = 0; i < numLevels; i++) {
            if (leafsAtLevelList[i].bitPattern() != 0) {
                lenOfShortCodeInTable.value = i;
                return;
            }
        }
    }
}



package thesis.compression.multi_tree3;


import java.util.*;
import java.io.*;
import org.omg.CORBA.IntHolder;
import thesis.compression.multi_tree.*;
import thesis.compression.multi_tree2.*;

/**
 *  An instance of this class decompresses a file compressed by
MultitreeCompressionManager3.
 */
public class MultitreeDecompressionManager3
extends MultitreeDecompressionManager2
{
    public static void main(String[] args) {

        String usage = new String("USAGE:  java
MulitreeDecompressionManager3 " +
            "<sourceFilename> <targetFilename>");

        try {
            if (args.length != 2) {
                throw new ArrayIndexOutOfBoundsException();
            }
            BitStringReader sourceFile = new BitStringReader(args[0]);
```

```
            BitStringWriter targetFile = new BitStringWriter(args[1]);
            new MultitreeDecompressionManager3(sourceFile, targetFile);
        } catch (ArrayIndexOutOfBoundsException wrongNumArgs) {
            System.out.println(usage);
        } catch (Exception e) {
            e.printStackTrace(new PrintStream(System.out));
        } finally {
            System.exit(1);
        }
    }

    public MultitreeDecompressionManager3(BitStringReader sourceFile,
                                          BitStringWriter targetFile)
    throws
    IOException
    {
        super(sourceFile, targetFile);
    }

    protected void buildDecodingMap(/* IN  */ BitStringReader
sourceFile,
                                    /* IN  */ int iBits,
                                    /* IN  */ int jBits,
                                    /* OUT */ HashMap decodingMap,
                                    /* OUT */ IntHolder
lenOfShortCodeInTable,
                                    /* IN  */ boolean isIWordMap)
    throws
    IOException
    {
        int numNonEmptyLevels =
(sourceFile.readBitString(5)).bitPattern();
        if (numNonEmptyLevels == 0) {
            return;
        }

        // build leafsAtLevelList for getHuffmanCodes method
        int firstNonEmptyLevel =
(sourceFile.readBitString(5)).bitPattern();
        int lenLevel = (sourceFile.readBitString(5)).bitPattern();
        int numLevels = numNonEmptyLevels + firstNonEmptyLevel;
        BitString[] leafsAtLevelList = new BitString[numLevels];
        for (int i = 0; i < numLevels; i++) {
            if (i < firstNonEmptyLevel) {
                leafsAtLevelList[i] = new BitString((short)lenLevel,
0);
            } else {
                leafsAtLevelList[i] =
sourceFile.readBitString(lenLevel);
            }
        }

        // getHuffmanCodes
        BitString[] freqOrderedHuffmanCodes =
VonNoymanNode.getHuffmanCodes(leafsAtLevelList);

        // fill decodingMap with (Huffman code --> jWord) mapping
```

```
        int bits2Read = isIWordMap ? iBits : jBits;
        int numLeafs = freqOrderedHuffmanCodes.length;
        for (int i = 0; i < numLeafs; i++) {
            decodingMap.put(freqOrderedHuffmanCodes[i],
sourceFile.readBitString(bits2Read));
        }

        // find length of shortest Huffman code
        for (int i = 0; i < numLevels; i++) {
            if (leafsAtLevelList[i].bitPattern() != 0) {
                lenOfShortCodeInTable.value = i;
                return;
            }
        }
    }

}



package thesis.compression.multi_tree4;


import java.util.*;
import java.io.*;
import org.omg.CORBA.IntHolder;
import thesis.compression.multi_tree.*;
import thesis.compression.multi_tree3.*;

/**
 *  An instance of this class decompresses a file compressed by
MultitreeCompressionManager4.
 */
public class MultitreeDecompressionManager4
extends MultitreeDecompressionManager3
{
    public static void main(String[] args) {

        String usage = new String("USAGE:  java
MulitreeDecompressionManager4 " +
            "<sourceFilename> <targetFilename>");

        try {
            if (args.length != 2) {
                throw new ArrayIndexOutOfBoundsException();
            }
            BitStringReader sourceFile = new BitStringReader(args[0]);
            BitStringWriter targetFile = new BitStringWriter(args[1]);
            new MultitreeDecompressionManager4(sourceFile, targetFile);
        } catch (ArrayIndexOutOfBoundsException wrongNumArgs) {
            System.out.println(usage);
        } catch (Exception e) {
            e.printStackTrace(new PrintStream(System.out));
        } finally {
            System.exit(1);
```

```
        }
    }

    public MultitreeDecompressionManager4(BitStringReader sourceFile,
                                          BitStringWriter targetFile)
    throws
    IOException
    {
        super(sourceFile, targetFile);
    }

    protected void decompressSourceFile(/* OUT */  BitStringWriter
targetFile,
                                        /* IN  */  BitStringReader
sourceFile,
                                        /* IN  */  int iBits,
                                        /* IN  */  int jBits,
                                        /* IN  */  int iTree,
                                        /* OUT */  HashMap[]
decodingMaps,
                                        /* IN  */  BitString remainder,
                                        /* IN  */  BitString offset)
    throws
    IOException
    {
        targetFile.writeBitString(offset);

        // build decoding maps (Huffman code ==> jWord)
        // remember length of shortest Huffman code in each map
        IntHolder[] lenOfShortHufCode = new IntHolder[iTree + 1];
        for (int i = 0; i < iTree; i++) {
            lenOfShortHufCode[i] = new IntHolder();
            buildDecodingMap(sourceFile, iBits, jBits, decodingMaps[i],
lenOfShortHufCode[i], i == iTree);
        }

        long numWordsToRead =
(sourceFile.readBitString(32)).bitPattern();
        sourceFile.setDecodeLimit(numWordsToRead);
        BitString iWord = null, jWord = null;


        // get first unencoded iWord in sourceFile
        iWord = sourceFile.readBitString(iBits);

        // read an unencoded iWord and an encoded jWord from sourceFile
        // write iWord and jWord to targetFile
        while (numWordsToRead > 0) {

            jWord =
sourceFile.decodeBitString(decodingMaps[iWord.bitPattern()],
lenOfShortHufCode[iWord.bitPattern()].value);
            targetFile.writeBitString(iWord);
            targetFile.writeBitString(jWord);
            numWordsToRead--;
            iWord = sourceFile.readBitString(iBits);
        }
```

```
        sourceFile.close();
        targetFile.writeBitString(remainder);
        targetFile.close();
    }
}
```

```
package thesis.compression.multi_tree5;

import java.util.*;
import java.io.*;
import org.omg.CORBA.IntHolder;
import thesis.compression.multi_tree.*;

/**
 *  An instance of this class decompresses a file compressed by
MultitreeCompressionManager5.
 */
public class MultitreeDecompressionManager5
extends
MultitreeDecompressionManager
{
    private BitStringWriter targetFile;
    private BitStringReader sourceFile;
    private HashMap[] decodingMaps;
    private int n;
    private BitString remainder;
    private BitString offset;


    public static void main(String[] args) {

        String usage = new String("USAGE:  java
MulitreeDecompressionManager5 " +
            "<sourceFilename> <targetFilename>");

        try {
            if (args.length != 2) {
                throw new ArrayIndexOutOfBoundsException();
            }
            BitStringReader sourceFile = new BitStringReader(args[0]);
            BitStringWriter targetFile = new BitStringWriter(args[1]);
            new MultitreeDecompressionManager5(sourceFile, targetFile);
        } catch (ArrayIndexOutOfBoundsException wrongNumArgs) {
            System.out.println(usage);
        } catch (Exception e) {
            e.printStackTrace(new PrintStream(System.out));
        } finally {
            System.exit(1);
        }
    }
```

```
    public MultitreeDecompressionManager5(BitStringReader sourceFile,
                                          BitStringWriter targetFile)
    throws
    IOException
    {
        super(sourceFile, targetFile);
    }

    protected void decompressSourceFile(/* OUT */  BitStringWriter
targetFile,
                                        /* IN  */  BitStringReader
sourceFile,
                                        /* IN  */  int n,
                                        /* OUT */  HashMap[]
decodingMaps,
                                        /* IN  */  BitString remainder,
                                        /* IN  */  BitString offset)
    throws
    IOException
    {
        targetFile.writeBitString(offset);

        // build decoding maps (Huffman code ==> class index)
        // remember length of shortest Huffman code in each map
        IntHolder[] lenOfShortHufCode = new IntHolder[n + 1];
        for (int i = 0; i <= n; i++) {
            lenOfShortHufCode[i] = new IntHolder();
            buildDecodingMap(sourceFile, n, decodingMaps[i],
lenOfShortHufCode[i], i == n);
        }

        // build decoding table (necklace class index ==> necklace
class)
        long numClasses = Necklace.numClasses(n);
        Assertion.assert(numClasses <= Integer.MAX_VALUE);
        int[] indexToClass = new int[(int)numClasses];
        Necklace.loadTable_indexToClass(n, indexToClass);

        long numBitStringsCompressed =
(sourceFile.readBitString(32)).bitPattern();
        sourceFile.setDecodeLimit(2L * numBitStringsCompressed);
        sourceFile.setN(n);
        BitString r = null, c = null;
        int rot = -1;

        // get first encoded rotation in sourceFile
        r = sourceFile.decodeBitString(decodingMaps[n],
lenOfShortHufCode[n].value);

        // read an encoded rotation and class index from sourceFile
        // write an unencoded bitstring to targetFile
        while (r.length() > 0) {

            rot = r.bitPattern();

            // c is class index
```

```
            c = sourceFile.decodeBitString(decodingMaps[0],
lenOfShortHufCode[0].value);

            // c is class
            c.reInit((short)n, indexToClass[c.bitPattern()]);

            // c is original unencoded bitstring !!!
            c.rShift(rot, true);

            // write c to the target file
            targetFile.writeBitString(c);

            // get next encoded rotation in sourceFile
            // a BitString of length 0 indicates end of sourceFile
            r = sourceFile.decodeBitString(decodingMaps[n],
lenOfShortHufCode[n].value);
        }
        sourceFile.close();
        targetFile.writeBitString(remainder);
        targetFile.close();
    }
}



package thesis.compression.multi_tree;



import java.util.*;
import java.lang.*;
import java.io.*;


/**
 *  This class contains all the supporting methods needed to implement
the RTA approach
 *  presented in the thesis.  See chapter 4 for an explanation of the r
and i values
 *  referred to thruout the code.
 */
public class Necklace {

    private final int[] bitStringToIndex;
    private final byte[] bitStringToRots;
    private final int[] indexToClass;
    private final int n;
    private final long numClasses;
    private final int indexLen;
    private final int rotLen;

    /** Returns the length in bits of the i associated with a symbol of
length n >= 1. */
    public static final int indexLength(int n) {
        return n - rotLength(n) + 1;
    }
```

```
    /** Returns the length in bits of the r associated with a symbol of
length n >= 1. */
    public static final int rotLength(int n) {
        return (int)Math.ceil(Math.log(n) / Math.log(2));
    }

    /** Helper function for numClasses(n) */
    public static int gcdEuclid(int a, int b) {
        if (b==0) {
            return a;
        } else {
            return gcdEuclid(b, a % b);
        }
    }

    /** Helper function for numClasses(n) */
    public static final boolean areRelativelyPrime(int a, int b) {
        return gcdEuclid(a, b) == 1 ? true : false;
    }

    /** Helper function for numClasses(n) */
    public static final int theta(int d) {
       int count = 0;
       for (int i = 1; i <= d; i++) {
          if (areRelativelyPrime(d, i)) {
              count++;
          }
       }
       return count;
    }

    /**
     *  Uses the Burnside formula to calculate the exact number of
classes for a given
     *  value of n.
     */
    public static final long numClasses(int n) {
        long sum = 0;
        for (int d = 1; d <= n; d++) {
            if (n % d == 0) {
                sum += theta(d) * Math.pow(2, n / d);
            }
        }
        return sum / n;
    }

    /**
     *  The necklace algorithm.  This method builds a table which
allows the efficient
     *  lookup of a necklace class based upon its index in a list of
all such classes.
     *
     *  PRECONDITION:   indexToClass.length == numClasses(n)
     *  POSTCONDITION:  indexToClass contains all the necklace classes
of length n
```

```
     *                  as given by the necklace algorithm described in
chapter 3 of the
     *                  thesis.
     */
    public static void loadTable_indexToClass( /* IN  */ int n,
                                               /* OUT */ int[]
indexToClass)
    {
        int k = 0;                                          //  index to
store next class at
        BitString c = new BitString((short)n, -1);          //  the first
necklace class
        indexToClass[k++] = c.bitPattern();                 //  store it at
index 0 of classes

        while (c.bitPattern() != 0) {
            int j = c.leastSig1();
            c.clearBit(j);                                  //  replace least
sig 1 with 0
            int jj = j - 1;
            int i = c.length() - 1;                         //  most
significant bit
            while (jj >= 0) {                               //  copy over,
copy over, ...
                c.assignBit(jj, c.bitAt(i));
                jj--;
                i--;
            }
            if ((n % (n - j)) == 0) {                       //  j check
                indexToClass[k++] = c.bitPattern();
            }
        }
    }

    /**
     *  This method builds two tables of size 2^n.  The
bitStringToIndex table allows
     *  the efficient lookup of the class mapped to by any bit string
of length n.  The
     *  bitStringToRots table allows the efficient lookup of the number
of rotations
     *  required to map any bit string of length n to its necklace
class.
     */
    public static void
    loadTables_bitStringToIndex_bitStringToRots(/* IN  */ int n,
                                                /* IN  */ int[]
indexToClass,
                                                /* IN  */ long
numClasses,
                                                /* OUT */ int[]
bitStringToIndex,
                                                /* OUT */ byte[]
bitStringToRots)
    {
        for (int i = 0; i < numClasses; i++) {
            BitString bs = new BitString(n, indexToClass[i]);
```

```
            int j = 0;
            do {
                bitStringToIndex[bs.bitPattern()] = i;
                bitStringToRots[bs.bitPattern()] = (byte)j;
                j++;
                bs.rShift(true);
            } while (bs.bitPattern() != indexToClass[i]);
        }
    }

    /**
     *  Constructs a Necklace object of length n.  This object provides
access to all
     *  the tables and methods needed to efficiently work with binary
necklaces of length n.
     */
    public Necklace(int n) {

        this.n = n;
        numClasses = numClasses(n);

        Assertion.assert(numClasses <= Integer.MAX_VALUE);
        indexToClass = new int[(int)numClasses];
        loadTable_indexToClass(n, indexToClass);

        bitStringToIndex = new int[(int)(Math.pow(2, n))];
        bitStringToRots = new byte[bitStringToIndex.length];
        loadTables_bitStringToIndex_bitStringToRots(
            n, indexToClass, numClasses, bitStringToIndex,
bitStringToRots);

        indexLen = indexLength(n);
        rotLen = rotLength(n);
    }

    /** Returns the index of the necklace class mapped to by bitString
*/
    public final int bitStringToIndex(int bitString) {
        return bitStringToIndex[bitString];
    }

    /** Returns the # of rotations required to map bitString to its
necklace class */
    public final byte bitStringToRots(int bitString) {
        return bitStringToRots[bitString];
    }

    /** Returns the necklace class at the given index. */
    public final int indexToClass(int index) {
        return indexToClass[index];
    }

    /** Returns the value of n being used by this Necklace object */
    public final int n() {
        return n;
    }
```

```
    /** Returns the number of classes associated with this Necklace
object */
    public final long numClasses() {
        return numClasses;
    }

    /** Returns the length in bits of the r's associated with this
Necklace object */
    public final int rotLength() {
        return rotLen;
    }

    /** Returns the length in bits of the i's associated with this
Necklace object */
    public final int indexLength() {
        return indexLen;
    }
}
```

```
package thesis.compression.multi_tree;


import java.util.*;
import java.io.*;
import thesis.compression.huffman.*;

/**
 *  Objects of this class are used to collect compression statistics
for any method that
 *  implements the StatsCollected Interface.  Each Stats object holds
statistics for
 *  one implementor of StatsCollected.
 */
public class Stats implements Comparable {

    private String filename;            // complete filename including
extension
    private String ext;                 // filename extension
    private int n;                      // word length used for
compression
    private int iBits;                  // num bits to specify index
of Huffman tree
    private int jBits;                  // num bits to specify leaf
node of Huffman tree
    private int iTree;                  // highest tree index (iWord
tree)
    private int lenOffset;              // # of uncompressed leading
bits in sourceFile

    private long sizeBefore;            // in bits
    private long sizeAfter;             // in bits
```

```
    private double[] stringDistr;        // % of source file mapped to
each Huffman tree
    private int[] classDistr;            // # classes mapped to each
Huffman tree

    private long sizeAfterHuf;           // file size achieved using
straight Huffman
    private long headerSize;             // header size


    public Stats(StatsCollected mcm)
    throws
    IOException,
    StatsNotAvailableException
    {
        n = mcm.n();
        iBits = mcm.iBits();
        jBits = mcm.jBits();
        iTree = mcm.iTree();
        lenOffset = mcm.lenOffset();
        filename = mcm.sourceFilename();

        int dotPos = filename.indexOf('.');
        ext = (dotPos != -1 ? filename.substring(dotPos) : "");

        mcm.buildFreqTables();

        sizeBefore = mcm.sourceFileSize();

        long numBitStrings = sizeBefore / n;  // int div

        stringDistr = new double[iTree];
        Arrays.fill(stringDistr, 0);
        Iterator it = mcm.lastTreeInspector();
        while (it.hasNext()) {
            VonNoymanNode node = (VonNoymanNode)it.next();
            stringDistr[node.index.bitPattern()] =
                (double)node.freq / (double)numBitStrings * 100d;
        }

        classDistr = mcm.treeSizes();

        mcm.compressSourceFile();

        sizeAfter = mcm.targetFileSize();
        headerSize = mcm.headerSize();
        mcm = null;
    }

    public Stats(StatsCollected mcm,
                 HuffmanCompressionManager hcm)
    throws
    IOException,
    StatsNotAvailableException
    {
        this(mcm);
        hcm.compress();
```

```
        sizeAfterHuf = hcm.sizeAfter();
        hcm = null;
    }

    public final int compareTo(Stats s) {
        if (ext.equals(s.ext)) {
            if (filename.equals(s.filename)) {
                if (n == s.n) {
                    if (iBits == s.iBits) {
                        return lenOffset - s.lenOffset;
                    } else {
                        return iBits - s.iBits;
                    }
                } else {
                    return n - s.n;
                }
            } else {
                return filename.compareTo(s.filename);
            }
        } else {
            return ext.compareTo(s.ext);
        }
    }

    public final int compareTo(Object obj) {
        return compareTo((Stats)obj);
    }

    public String toString() {
        StringBuffer sb = new StringBuffer();
        String nl = new String("\r\n");

        sb.append("FILENAME : ");
        sb.append(filename);
        sb.append(nl);

        sb.append("n        : ");
        sb.append(n);
        sb.append(nl);

        sb.append("iBits    : ");
        sb.append(iBits);
        sb.append(nl);

        sb.append("jBits    : ");
        sb.append(jBits);
        sb.append(nl);

        sb.append("offset   : ");
        sb.append(lenOffset);
        sb.append(nl);
        sb.append(nl);

        sb.append("file size: ");
        sb.append(toKB(sizeBefore));
        sb.append(nl);
        sb.append(nl);
```

```
        sb.append("header size: ");
        sb.append(headerSize / 8);
        sb.append(" bytes");
        sb.append(nl);
        sb.append(nl);

        if (sizeAfterHuf > 0) {
            sb.append("HUF size : ");
            sb.append(toKB(sizeAfterHuf));
            sb.append(nl);

            sb.append("change   : ");
            sb.append(change(sizeBefore, sizeAfterHuf));
            sb.append(nl);
            sb.append(nl);
        }

        sb.append("MTC size : ");
        sb.append(toKB(sizeAfter));
        sb.append(nl);

        sb.append("change   : ");
        sb.append(change(sizeBefore, sizeAfter));
        sb.append(nl);
        sb.append(nl);

        sb.append("tree    strings    leafs");
        sb.append(nl);
        double tmp;
        for (int i = 0; i < iTree; i++) {
            sb.append(i);
            sb.append((i < 10 ? "       " : "      "));
            tmp = round(stringDistr[i], 0.1);
            sb.append(tmp);
            sb.append("%");
            sb.append((tmp < 10 ? "       " : "      "));
            sb.append(classDistr[i]);
            sb.append(nl);
        }

        return sb.toString();
    }

    private final String change(long orig, long cur) {
       long diff = cur - orig;
       return String.valueOf(round((double)diff / (double)orig * 100d,
0.1d)) + "%";
    }

    private final String toKB(long bits) {
        return String.valueOf(round((double)bits / 8d / 1024d, 0.001d))
+ " KB";
    }

    private final double round(double number,
                               double placeValue ) {
```

```
        double recipPlaceValue = 1.0/placeValue;
        return ( (int)(number*recipPlaceValue+0.5d) / recipPlaceValue
);
    }

}
```

```
package thesis.compression.multi_tree;

import java.util.*;
import java.io.*;

/**
 *  This interface declares methods needed to collect statistics on the
various
 *  compression classes like HuffmanCompressionManager and
MultitreeCompressionManager).
 *  Classes which implement this interface can be analyzed by the
StatsManager.
 */
public interface StatsCollected
{
    public int iBits();
    public int jBits();
    public int iTree();
    public int n();
    public int lenOffset();
    public String sourceFilename();
    public long sourceFileSize() throws StatsNotAvailableException;
    public long targetFileSize() throws StatsNotAvailableException;
    public Iterator lastTreeInspector() throws
StatsNotAvailableException;
    public long headerSize() throws StatsNotAvailableException;
    public int[] treeSizes() throws StatsNotAvailableException;

    public void buildFreqTables() throws IOException;
    public void compressSourceFile() throws IOException;
}
```

```
package thesis.compression.multi_tree;

import java.util.*;
import java.io.*;
import java.text.*;
import thesis.compression.huffman.*;
import thesis.compression.huffman2.*;
import thesis.compression.multi_tree2.*;
import thesis.compression.multi_tree3.*;
import thesis.compression.multi_tree4.*;
import thesis.compression.multi_tree5.*;
```

```
/**
 *  This static class drives a simple text based menu which allows
testing of all
 *  implemented compression algorithms.  The main method of this class
is the standard
 *  entry point into all the compression algorithms created for this
thesis.
 */
public class StatsManager {

    public static final String szOutDir = "c:\\My
Documents\\compressedFiles\\";
    public static final String szInDir  = "c:\\My
Documents\\filesToCompress\\";
    private static TreeSet registeredStats = new TreeSet();


    public static void main(String[] args) throws Exception {


        System.out.println("\nMultitree Compression Tool, May 6,
2001\n");
        System.out.println("USAGE");
        System.out.println("1) Create the directory '" + szInDir +
"'.");
        System.out.println("2) Put all files to compress in specified
directory.");
        System.out.println("3) Follow on screen prompts.");
        System.out.println("4) Compressed files & date stamped stats
file at " + szOutDir);

        BufferedReader inUser = new BufferedReader(new
InputStreamReader(System.in));
        System.out.println("\nselect an MTC method");
        System.out.println("1) rotational trees");
        System.out.println("2) rotational trees (tree 0 & N only)");
        System.out.println("3) indexed trees");
        System.out.println("4) indexed trees + tight header scheme");
        System.out.println("5) indexed trees + tight header scheme -
Huffman tree for words");
        System.out.print("Selection: ");
        int method = Integer.parseInt(inUser.readLine());
        System.out.println();

        File outDir = new File(szOutDir);
        if (!outDir.exists() && !outDir.mkdir()) {
            throw new IOException("Can't create " + szOutDir +
"!\nCreate the directory " +
                "manually or change file permissions then restart this
tool.\n");
        }
        File inDir = new File(szInDir);
        if (!inDir.exists())  {
            throw new FileNotFoundException(
                "Input directory " + szInDir + " does not exist!\n");
        }
        String[] sourceFiles = inDir.list();
```

```
        if (sourceFiles == null || sourceFiles.length == 0) {
            throw new FileNotFoundException(
                "No source files found in '" + szInDir + "'!!!!");
        }

        String[] targetFiles = new String[sourceFiles.length];
        for (int i = 0; i < targetFiles.length; i++) {
            int dotPos = sourceFiles[i].indexOf('.');
            if (dotPos > 0) {
                targetFiles[i] = sourceFiles[i].substring(0, dotPos);
            } else {
                targetFiles[i] = sourceFiles[i];
            }
        }

        switch (method) {
            case 1:

            case 2:     rotationalTrees(inUser, outDir, inDir,
sourceFiles, targetFiles, method);
                        break;
            case 3:

            case 4:

            case 5:     indexedTrees(inUser, outDir, inDir,
sourceFiles, targetFiles, method);
                        break;
            default:    throw new Exception("\nInvalid selection -
restart the tool and try again.");
        }
    }

    private static void rotationalTrees(BufferedReader inUser,
                                        File outDir,
                                        File inDir,
                                        String[] sourceFiles,
                                        String[] targetFiles,
                                        int mtcMethod)
    throws
    Exception
    {

        System.out.print("Enter a lower bound for N:  ");
        int lowN = Integer.parseInt(inUser.readLine());
        System.out.print("Enter an upper bound for N:  ");
        int highN = Integer.parseInt(inUser.readLine());

        boolean offsetsOn = false;
        System.out.print("Turn offsets ON (y/n)? ");
        if (inUser.readLine().equalsIgnoreCase("y")) {
            offsetsOn = true;
        }

        boolean hufCompOn = false;
        System.out.print("Turn Huffman comparisons ON (y/n)? ");
        if (inUser.readLine().equalsIgnoreCase("y")) {
```

```
            hufCompOn = true;
        }

        System.out.print("\nTable building memory requirements for this
run: ");
        System.out.print((long)(5 * Math.pow(2, highN) + 4 *
Necklace.numClasses(highN)));
        System.out.println(" bytes");

        System.out.println("\nPROGRESS:");
        for (int i = 0; i < sourceFiles.length * (highN - lowN + 1);
i++) {
            System.out.print(".");
        }
        System.out.println();

        for (int i = 0; i < sourceFiles.length; i++) {
            for (int n = lowN; n <= highN; n++) {
                for (int offsetLen = 0; offsetLen == 0 || (offsetsOn &&
offsetLen < n); offsetLen++) {

                    BitStringReader bsr = new BitStringReader(szInDir +
sourceFiles[i], n);
                    BitStringWriter bsw = new BitStringWriter(
                        szOutDir + targetFiles[i] + n + "_" + offsetLen
+ ".MTC" +
                        (mtcMethod == 1 ? "1" : "5"));

                    StatsCollected mcm;
                    if (mtcMethod == 1) {
                        mcm = new MultitreeCompressionManager(bsr, bsw,
n, offsetLen);
                    } else {
                        mcm = new MultitreeCompressionManager5(bsr,
bsw, n, offsetLen);
                    }

                    if (hufCompOn) {
                        BitStringReader bsr2 = new
BitStringReader(szInDir + sourceFiles[i], n);
                        BitStringWriter bsw2 = new BitStringWriter(
                            szOutDir + targetFiles[i] + n + "_" +
offsetLen + ".HUF");
                        HuffmanCompressionManager hcm =
                            new HuffmanCompressionManager(bsr2, bsw2,
n, offsetLen);
                        registerStats(new Stats(mcm, hcm));
                    } else {
                        registerStats(new Stats(mcm));
                    }
                }
                System.out.print(".");
            }
        }
        System.out.println("\n");
        outputStats(mtcMethod + " (rotational trees)");
    }
```

```
    private static void indexedTrees(BufferedReader inUser,
                                     File outDir,
                                     File inDir,
                                     String[] sourceFiles,
                                     String[] targetFiles,
                                     int mtcMethod)
    throws
    Exception
    {
        String userPair = null;
        int indexOfSpace = -1, userI = -1, userJ = -1, lowN =
BitString.MAXLEN;
        ArrayList arrayI = new ArrayList(), arrayJ = new ArrayList();

        System.out.print("Enter a space delimited iBits jBits pair
(like 4 12 or 8 8): ");
        userPair = inUser.readLine();
        while (userPair != null && userPair.length() >= 3) {
            indexOfSpace = userPair.indexOf(' ');
            userI = Integer.parseInt(userPair.substring(0,
indexOfSpace));
            userJ = Integer.parseInt(userPair.substring(indexOfSpace +
1));
            arrayI.add(new Integer(userI));
            arrayJ.add(new Integer(userJ));
            if (lowN > userI + userJ) {
                lowN = userI + userJ;
            }
            System.out.print("Enter another pair or <RETURN> to begin
compression: ");
            userPair = inUser.readLine();
        }

        System.out.print("Enter an offset bound (0 to " + (lowN - 1) +
"): ");
        int offsetBound = Integer.parseInt(inUser.readLine());
        if (offsetBound < 0) {
            offsetBound = 0;
        } else if (offsetBound > lowN - 1) {
            offsetBound = lowN - 1;
        }

        boolean hufCompOn = false;
        System.out.print("Turn Huffman comparisons ON (y/n)? ");
        if (inUser.readLine().equalsIgnoreCase("y")) {
            hufCompOn = true;
        }

        System.out.println("\nPROGRESS:");
        for (int a = 0; a < sourceFiles.length * arrayI.size() *
(offsetBound + 1); a++) {
            System.out.print(".");
        }
        System.out.println();
```

```
        for (int a = 0; a < sourceFiles.length; a++) {
            for (int b = 0; b < arrayI.size(); b++) {
                for (int c = 0; c <= offsetBound; c++) {

                    userI = ((Integer)arrayI.get(b)).intValue();
                    userJ = ((Integer)arrayJ.get(b)).intValue();
                    BitStringReader bsr = new BitStringReader(szInDir +
sourceFiles[a]);
                    BitStringWriter bsw = new BitStringWriter(
                        szOutDir + targetFiles[a] + userI + "_" + userJ
+ "_" + c + ".MTC" +
                        (mtcMethod - 1));

                    StatsCollected mcm;
                    if (mtcMethod == 3) {
                        mcm = new MultitreeCompressionManager2(bsr,
bsw, userI, userJ, c);
                    } else if (mtcMethod == 4) {
                        mcm = new MultitreeCompressionManager3(bsr,
bsw, userI, userJ, c);
                    } else {
                        mcm = new MultitreeCompressionManager4(bsr,
bsw, userI, userJ, c);
                    }

                    if (hufCompOn) {
                        int n = userI + userJ;
                        BitStringReader bsr2 = new
BitStringReader(szInDir + sourceFiles[a], n);
                        BitStringWriter bsw2 = new BitStringWriter(
                            szOutDir + targetFiles[a] + n + "_" + c +
".HUF");
                        HuffmanCompressionManager2 hcm2 =
                            new HuffmanCompressionManager2(bsr2, bsw2,
n, c);
                        registerStats(new Stats(mcm, hcm2));
                    } else {
                        registerStats(new Stats(mcm));
                    }

                    System.out.print(".");
                }
            }
        }
        System.out.println("\n");
        outputStats(mtcMethod + " (indexed trees)");
    }

    public static final void registerStats(Stats stats) {
        registeredStats.add(stats);
    }

    public static void outputStats(String method)
    throws
    FileNotFoundException
    {
```

```
        SimpleDateFormat formatter = new
SimpleDateFormat("D_hh_mm_ss");
        Date date = new Date();
        String szDate = formatter.format(date);
        PrintStream statsFile = new PrintStream(
            new FileOutputStream(szOutDir + "stats" + szDate +
".txt"));
        PrintStream out = System.out;

        for (int i = 0; i < 2; i++) {
            System.setOut(i == 0 ? out : statsFile);
            System.out.println("Compression method: " + method);
            System.out.println("Compressions performed: " +
registeredStats.size());
            System.out.println();
            Iterator it = registeredStats.iterator();
            while (it.hasNext()) {
                System.out.println(it.next());
                System.out.println("--------------------------");
                System.out.println();
            }
        }
        statsFile.close();
        System.setOut(out);
    }
}
```

```
package thesis.compression.multi_tree;


public class StatsNotAvailableException extends Exception {
    public StatsNotAvailableException() {
        super("StatsCollected method calls out of sequence");
    }

    public StatsNotAvailableException(String str) {
        super(str);
    }

}
```

```
package thesis.compression.multi_tree;

import java.util.*;
import java.io.*;

/**
 *  Poorly named class which implements the Neuman (not John VonNoyman)
technique of
```

```
 *  building a Huffman tree.  The technique represents the leaves of a
Huffman tree using
 *  a decimal number.  For example:  0.0235 indicates that the tree has
0 leaves at level
 *  0, 0 leaves at level 1, 2 leaves at level 2, 3 leaves at level 3,
and 5 leaves at level
 *  4.  Using this technique a Huffman tree is built from a frequency
table by a
 *  series of decimal shift and addition opperations which seems more
convenient than the
 *  actual construction of a binary tree.
 */
public class VonNoymanNode implements Comparable {

    public long freq;
    public int[] leafsAtLevel;
    public BitString index;

    /**
     *  Returns the decimal digits that represent the leaves at each
level of the
     *  Huffman tree built from the frequency table represented by
freqOrderedVNN.
     *  The decimal digits are returned as an array of integers as each
digit may
     *  exceed 9 depending upon the size of the resultant Huffman tree.
     *
     *  PRECONDITION:  freqOrderedVNN is not empty
     *  POSTCONDITION: freqOrderedVNN is trashed
     */
    public static int[] vonNoymanAlgorithmInt( /* IN/OUT */ TreeSet
freqOrderedVNN)
    throws
    NoSuchElementException
    {
        // special case
        if (freqOrderedVNN.size() == 1) {
            return new int[]  {0, 1};
        }

        // PLC:  vnn1.leafsAtLevel contains the # of leafs at each
level of Huffman tree
        VonNoymanNode vnn1, vnn2;
        do {
            vnn1 = (VonNoymanNode)freqOrderedVNN.first();
            freqOrderedVNN.remove(vnn1);
            if (freqOrderedVNN.isEmpty()) break;
            vnn2 = (VonNoymanNode)freqOrderedVNN.first();
            freqOrderedVNN.remove(vnn2);
            vnn1.add(vnn2);
            freqOrderedVNN.add(vnn1);
        } while (true);
        return vnn1.leafsAtLevel;
    }

    /**
```

```
     *  Identical to vonNoymanAlgorithmInt(freqOrderedVNN) except that
the resultant
     *  list of decimal digits is returned as a BitString[] instead of
an int[].
     */
    public static final BitString[] vonNoymanAlgorithm( /* IN/OUT */
TreeSet freqOrderedVNN)
    throws
    NoSuchElementException
    {
        int[] leafsAtLevel = vonNoymanAlgorithmInt(freqOrderedVNN);
        BitString[] retValue = BitString.toBitStringArr(leafsAtLevel);
        Assertion.assert(leafsAtLevel.length == retValue.length);
        return retValue;
    }

    /**
     *  Returns an array of BitStrings which represents the Huffman
codes of a Huffman
     *  tree built using leafsAtLevelList.  The codes are ordered from
shortest to
     *  longest.
     */
    public static BitString[] getHuffmanCodes(BitString[]
leafsAtLevelList) {

        // need one Huffman code per leaf so count leaves
        int numCodesNeeded = 0;
        for (int i = 0; i < leafsAtLevelList.length; i++) {
            numCodesNeeded += leafsAtLevelList[i].bitPattern();
        }

        BitString[] retValue = new BitString[numCodesNeeded];
        int pos = 0;
        int value = 0;

        for (int i = 0; i < leafsAtLevelList.length; i++) {
// level
            for (int j = 0; j < leafsAtLevelList[i].bitPattern(); j++)
{  // leaf
                retValue[numCodesNeeded - 1 - pos] = new
BitString((short)(i), value);
                pos++;
                value++;
            }
            value <<= 1;
        }

        return retValue;
    }

    /** Constructor */
    public VonNoymanNode(BitString index) {
        freq = 1;
        leafsAtLevel =  new int[] {1};
        this.index = index;
    }
```

```
    /**
     *  Helper method for VonNoymanAlgorithm methods.
     */
    public void add(VonNoymanNode rValue) {

        // add freq attributes
        freq += rValue.freq;

        // add leafsAtLevel attributes
        int[] longer, shorter;
        if (leafsAtLevel.length > rValue.leafsAtLevel.length) {
            longer = leafsAtLevel;
            shorter = rValue.leafsAtLevel;
        } else {
            longer = rValue.leafsAtLevel;
            shorter = leafsAtLevel;
        }

        int[] temp = new int[longer.length + 1];
        temp[0] = 0;
        for (int i = 0; i < longer.length; i++) {
            temp[i + 1] = longer[i] + (i < shorter.length ? shorter[i]
: 0);
        }
        leafsAtLevel = temp;

        // ignore classIndex attribute for this operation
    }

    public final void incrFreq() {
        freq++;
    }

    public final String toString() {
        return index.toString() + "\t(" + freq + ")";
    }

    public final boolean equals(Object obj) {
        if ((obj != null) && (obj instanceof VonNoymanNode)) {
            VonNoymanNode vnn = (VonNoymanNode)obj;
            return freq == vnn.freq && index.equals(vnn.index);
        }
        return false;
    }

    public final int compareTo(VonNoymanNode vnn) {
        long diff = freq - vnn.freq;
        if (diff > 0) {
            return 1;
        } else if (diff < 0) {
            return -1;
        } else {
            return index.compareTo(vnn.index);
        }
    }
```

```
    public final int compareTo(Object obj) {
        return compareTo((VonNoymanNode)obj);
    }
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

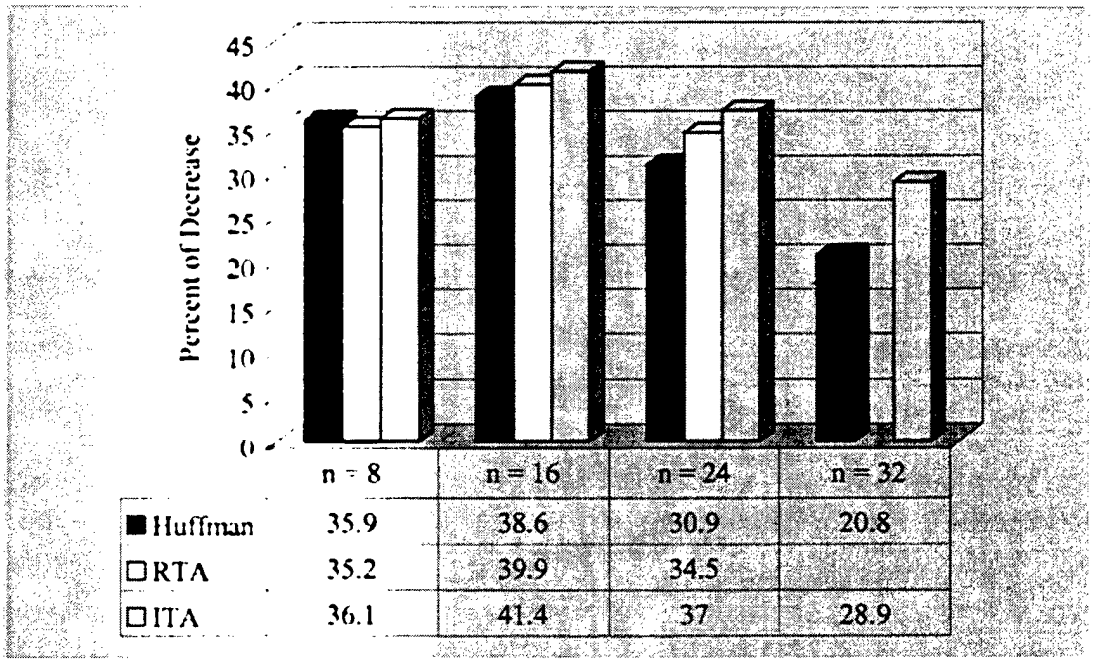
APPENDIX B: SUMMARY OF EMPIRICAL DATA

To determine the performance of both RTA and ITA, we ran comparisons against standard Huffman encoding, using the Canterbury Corpus [19] and the well-known "Lenna" photograph [20]. The corpus is a suite of eleven text-based files, commonly used as an industry benchmark for testing lossless compression algorithms. This collection was developed in 1997 as an improved version of the older Calgary corpus. The files were chosen because their results on existing compression algorithms are "typical", and so it is hoped that they will be representative for new methods of compression as well. Lena is a Tagged Image File (TIF), which is an industry benchmark for image compression. The files that comprise The Canterbury Corpus are:

file	Abbrev	Category	Size in bytes
alice29.txt	text	English text	152089
asyoulik.txt	play	Shakespeare	125179
cp.html	html	HTML source	24603
fields.c	Csrc	C source	11150
grammar.lsp	list	LISP source	3721
kennedy.xls	Excl	Excel Spreadsheet	1029744
lcet10.txt	tech	Technical writing	426754
plrabn12.txt	poem	Poetry	481861
ptt5	fax	CCITT test set	513216
sum	SPRC	SPARC Executable	38240
xargs.1	man	GNU manual page	4227

Table 15: Canterbury Corpus Test Suite

The following graphs summarize the results of RTA, ITA, and Huffman encoding against the test suite files mentioned above. The title of the graph identifies the file and the tabular data represents the optimal compression for the given n. The ITA results of each graph represent the optimal combination of |iBits| + |jBits| = n for each file. The RTA results are do not extend beyond n = 24 due to the exponential memory requirements of this approach.



	n = 8	n = 16	n = 24	n = 32
Huffman	35.9	38.6	30.9	20.8
RTA	35.2	39.9	34.5	
ITA	36.1	41.4	37	28.9

Chart 2: Compression Results for Test File Fields.c

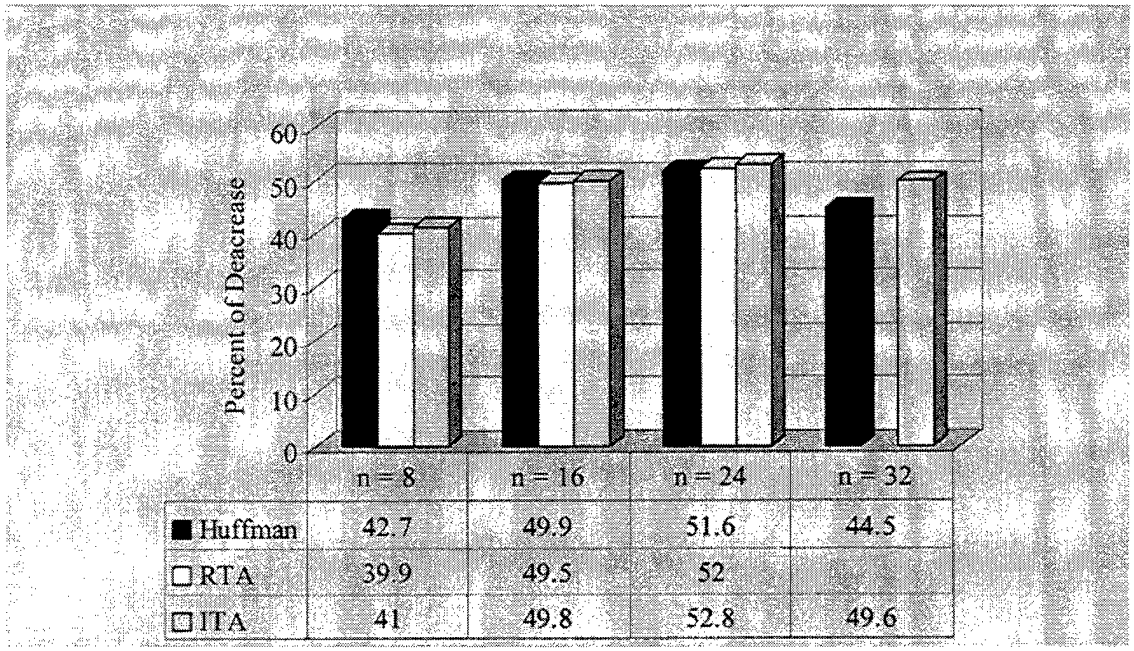


Chart 3: Compression Results for Test File plrabn.txt

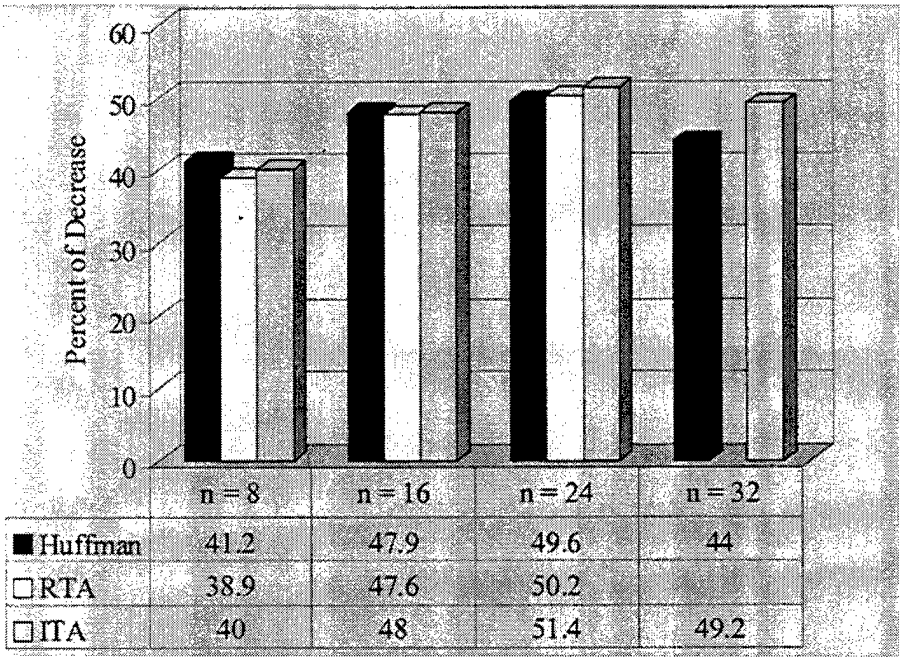
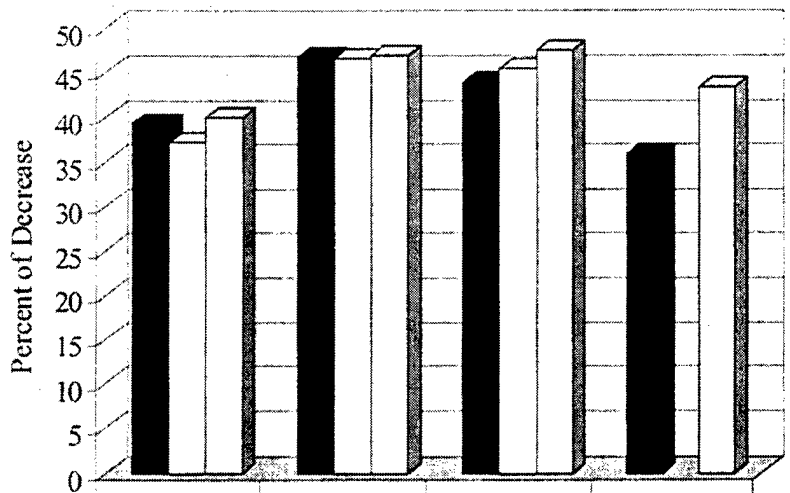
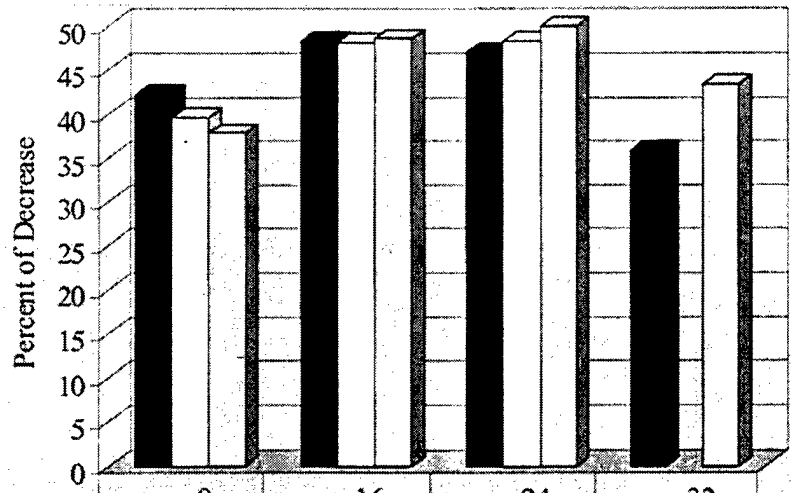


Chart 4: Compression Results for Test File icet.txt



	n = 8	n = 16	n = 24	n = 32
■ Huffman	39.3	46.7	43.9	35.8
□ RTA	37.2	46.5	45.4	
□ ITA	40	46.9	47.5	43.4

Chart 5: Compression Results for Test File asyoulik.txt



	n = 8	n = 16	n = 24	n = 32
■ Huffman	42.2	48.3	47	35.8
□ RTA	39.6	48	48.2	
□ ITA	37.8	48.5	49.9	43.4

Chart 6: Compression Results for Test File alice29.txt

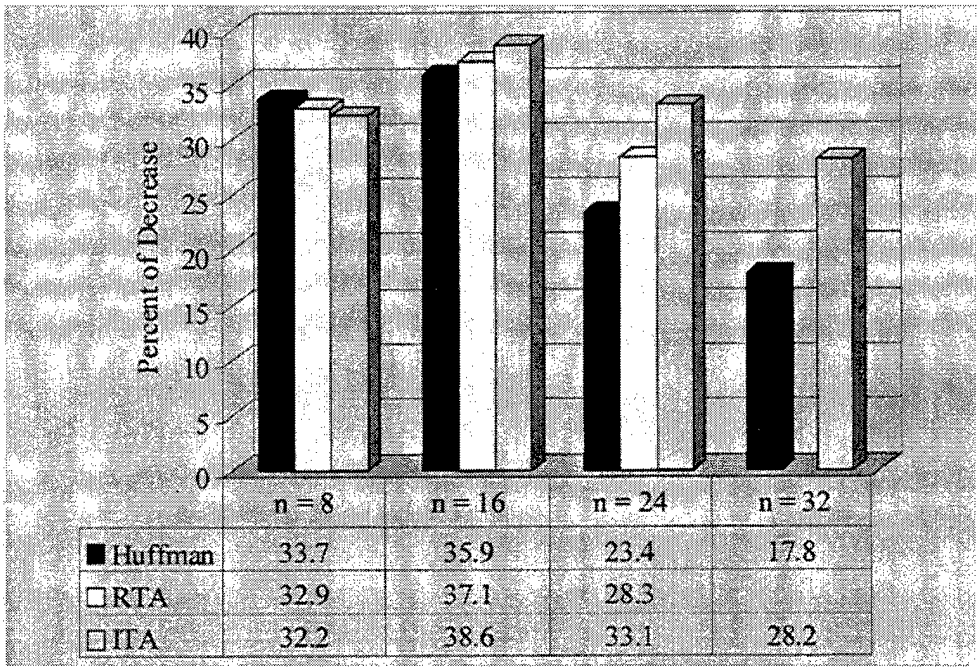


Chart 7: Compression Results for Test File cp.html

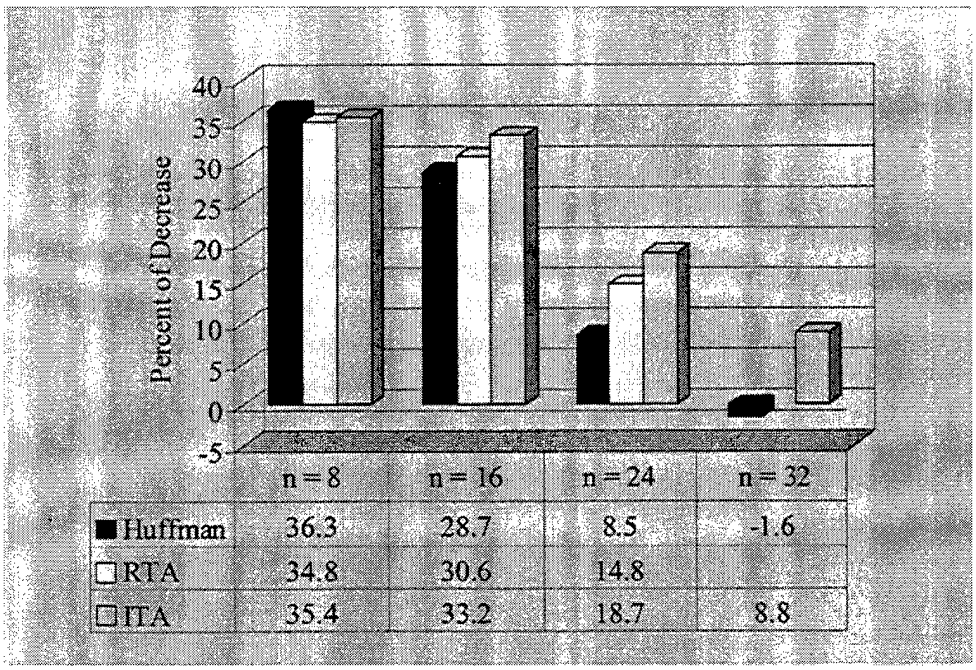
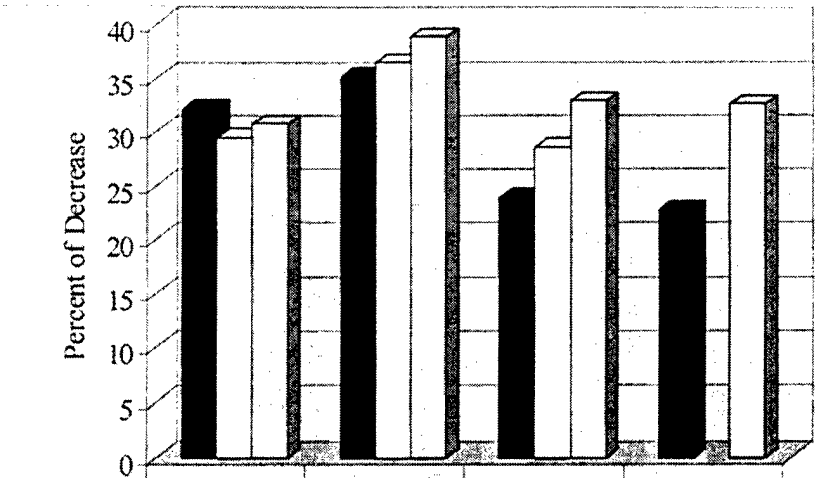
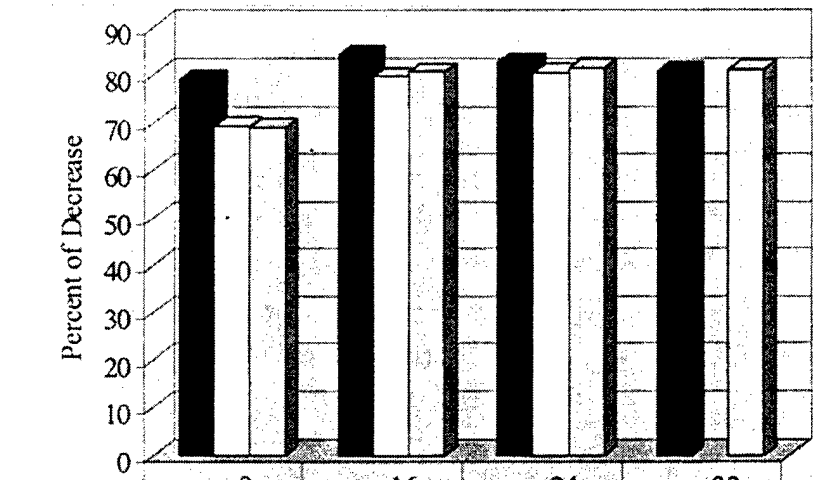


Chart 8: Compression Results for Test File xargs.1



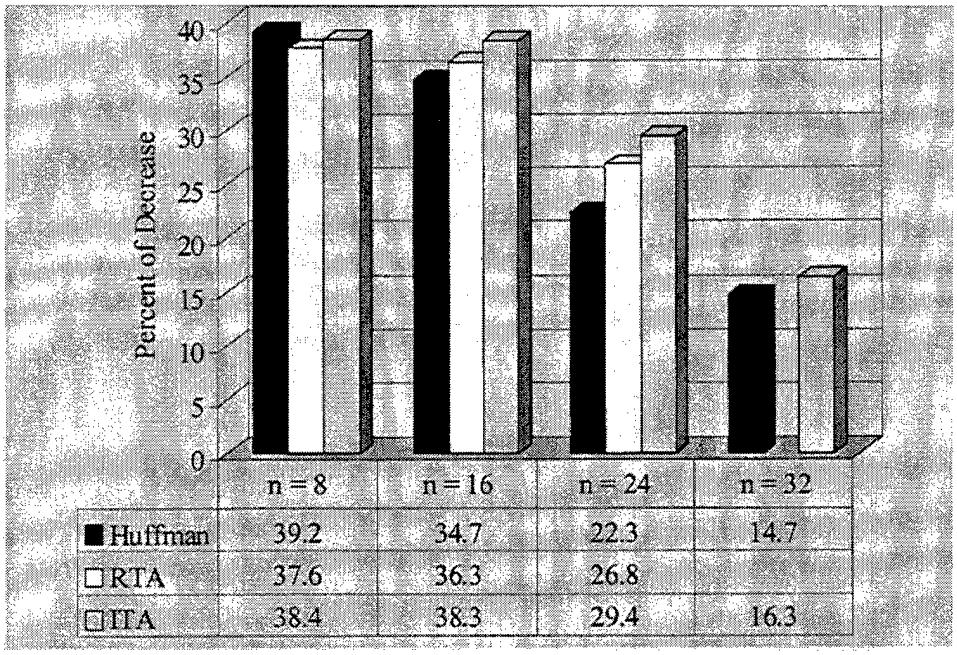
	n = 8	n = 16	n = 24	n = 32
■ Huffman	32.1	35.1	23.8	22.7
□ RTA	29.4	36.4	28.5	
□ ITA	30.7	38.8	32.8	32.5

Chart 9: Compression Results for Test File sum



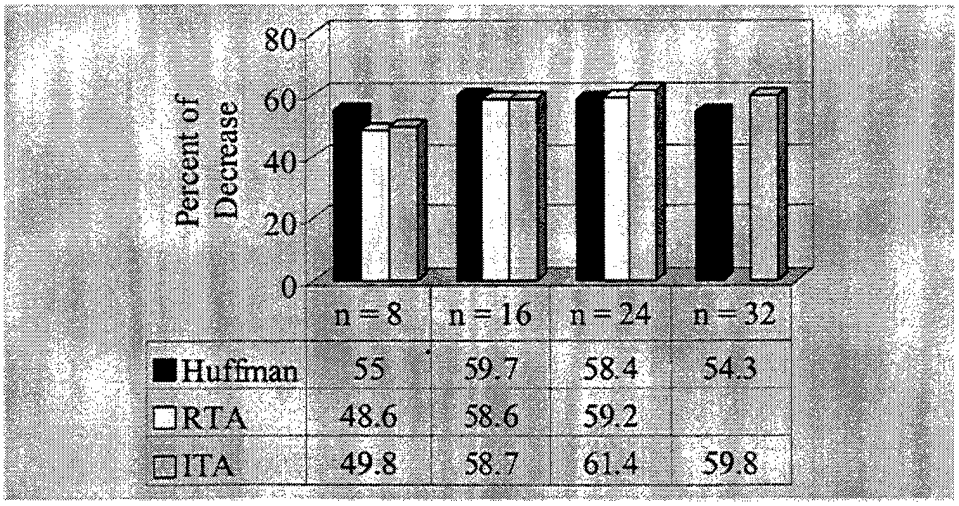
	n = 8	n = 16	n = 24	n = 32
■ Huffman	79.1	84.1	82.6	80.7
□ RTA	69.1	79.5	80.3	
□ ITA	69	80.6	81.4	81

Chart 10: Compression Results for Test File ptt5



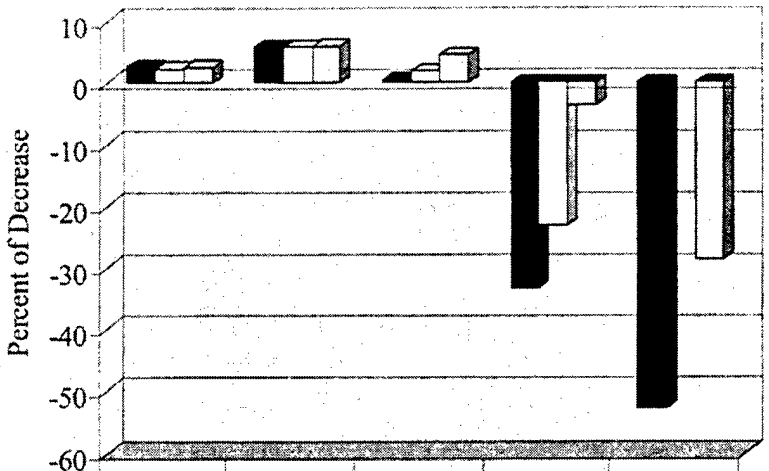
	n = 8	n = 16	n = 24	n = 32
Huffman	39.2	34.7	22.3	14.7
RTA	37.6	36.3	26.8	
ITA	38.4	38.3	29.4	16.3

Chart 11: Compression Results for Test File grammer.lsp



	n = 8	n = 16	n = 24	n = 32
Huffman	55	59.7	58.4	54.3
RTA	48.6	58.6	59.2	
ITA	49.8	58.7	61.4	59.8

Chart 12: Compression Results for Test File kennedy.xls



	n = 8	n = 12	n = 16	n = 24	n = 32
■Huffman	2.6	5.8	0.3	-33.2	-52.7
□RTA	2.2	5.7	1.8	-23.1	
□ITA	2.5	5.9	4.4	-3.7	-28.5

Chart 13: Compression Results for Test File lena.tif

APPENDIX C: MAPPING OF ASCII KEYBOARD CHARACTERS BY RTA

Character	ASCII Value	Binary	Index Tree
space	32	00100000	2
!	33	00100001	7
"	34	00100010	6
#	35	00100011	6
$	36	00100100	2
%	37	00100101	6
&	38	00100110	6
	39	00100111	5
(	40	00101000	2
)	41	00101001	2
*	42	00101010	2
+	43	00101011	6
,	44	00101100	4
-	45	00101101	4
.	46	00101110	4
/	47	00101111	4
0	48	00110000	2
1	49	00110001	2
2	50	00110010	2
3	51	00110011	2
4	52	00110100	2
5	53	00110101	2
6	54	00110110	2
7	55	00110111	5
8	56	00111000	2
9	57	00111001	2
:	58	00111010	2
;	59	00111011	2
<	60	00111100	2
=	61	00111101	2
>	62	00111110	2
?	63	00111111	2
@	64	01000000	1
A	65	01000001	1
B	66	01000010	1
C	67	01000011	6
D	68	01000100	1
E	69	01000101	5
F	70	01000110	5
G	71	01000111	5
H	72	01001000	1
I	73	01001001	7
J	74	01001010	4
K	75	01001011	6
L	76	01001100	4
M	77	01001101	4
N	78	01001110	4
O	79	01001111	4

P	80	01010000	1
Q	81	01010001	1
R	82	01010010	1
S	83	01010011	6
T	84	01010100	1
U	85	01010101	1
V	86	01010110	5
W	87	01010111	5
X	88	01011000	3
Y	89	01011001	3
Z	90	01011010	3
[	91	01011011	3
\	92	01011100	3
]	93	01011101	3
^	94	01011110	3
_	95	01011111	3
`	96	01100000	1
a	97	01100001	1
b	98	01100010	1
c	99	01100011	6
d	100	01100100	1
e	101	01100101	1
f	102	01100110	1
g	103	01100111	5
h	104	01101000	1
I	105	01101001	1
j	106	01101010	1
k	107	01101011	6
l	108	01101100	1
m	109	01101101	1
n	110	01101110	4
o	111	01101111	4
p	112	01110000	1
q	113	01110001	1
r	114	01110010	1
s	115	01110011	1
t	116	01110100	1
u	117	01110101	1
v	118	01110110	1
w	119	01110111	1
x	120	01111000	1
y	121	01111001	1
z	122	01111010	1
{	123	01111011	1
|	124	01111100	1
}	125	01111101	1
~	126	01111110	1

Table 15: ASCII Mapping by RTA

APPENDIX D: SAMPLE OUTPUT FROM ITA COMPRESSION PROGRAM

Compression method: 4 (indexed trees)

Compressions performed: 1

FILENAME : c:\My Documents\filesToCompress\sum

n : 16

iBits : 4

jBits : 12

offset : 0

file size: 37.344 KB

header size: 3719 bytes

HUF size : 24.199 KB

change : -35.1%

MTC size : 23.258 KB

change : -37.6%

tree	strings	leafs
0	39.0%	892
1	2.5%	61
2	6.7%	222
3	16.0%	221
4	4.1%	175
5	2.7%	110
6	13.0%	282
7	8.5%	166
8	2.9%	33
9	1.5%	47
10	0.5%	46
11	0.3%	17
12	0.3%	13
13	1.2%	30
14	0.4%	18
15	0.5%	56

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX E: RTA COMPRESSION FORMAT

```
n                                                                    5  bits

# of offset (uncompressed) bits f, where f < n                       5  bits
actual offset bits                                                   f  bits

# of remainder (uncompressed) bits r, where r < n                    5  bits
actual remainder bits                                                r  bits

class tree 0
-------------------------------------------------------------------------
bits used b to represent # leafs L at each level of Huffman tree     5  bits
# of levels v in Huffman tree                                        5  bits
leafs in level v - 1 of Huffman tree  (least freq)                   b  bits
leafs in level v - 2 of Huffman tree                                 b  bits
...
leafs in level 0 of Huffman tree      (most freq)                    b  bits

level v - 1 class(es)                 (longest code)    ceil(log2n) bits
level v - 2 class(es)                                   ceil(log2n) bits
...
level 0 class(es)                     (shortest code)   ceil(log2n) bits
-------------------------------------------------------------------------

class tree 1
-------------------------------------------------------------------------
(same as above)
-------------------------------------------------------------------------
.
.
.
class tree n - 1
-------------------------------------------------------------------------
(same as above)
-------------------------------------------------------------------------

rotation tree n
-------------------------------------------------------------------------
bits used b to represent # leafs f at each level of Huffman tree     5  bits
# of levels v in Huffman tree                                        5  bits
leafs in level v - 1 of Huffman tree  (least freq)                   b  bits
leafs in level v - 2 of Huffman tree                                 b  bits
...
leafs in level 0 of Huffman tree      (most freq)                    b  bits

level v - 1 rotation(s)               (longest code)    ceil(log2n) bits
level v - 2 rotation(s)                                 ceil(log2n) bits
...
level 0 rotation(s)                   (shortest code)   ceil(log2n) bits
-------------------------------------------------------------------------
```

# bitstrings processed	32 bits
encoded rotations	variable
encoded class index	variable
encoded rotations	variable
encoded class index	variable
...	
last encoded rotations	variable
last encoded class index	variable

APPENDIX F: ITA COMPRESSION FORMAT

```
iBits                                                                    5  bits
jBits                                                                    5  bits

# of offset (uncompressed) bits f, where f < n                           5  bits
actual offset bits                                                       f  bits

# of remainder (uncompressed) bits r, where r < n                        5  bits
actual remainder bits                                                    r  bits

index tree 0
-------------------------------------------------------------------------------
# of non-empty levels v in Huffman tree                                  5  bits
first non-empty level F                                                  5  bits
bits used b to represent # leafs L at each level of Huffman tree         5  bits
leafs in level F + v - 1 of Huffman tree  (least freq)                   b  bits
leafs in level F + v - 2 of Huffman tree                                 b  bits
...
leafs in level F of Huffman tree          (most freq)                    b  bits

level F + v - 1 jWord                     (longest code)            jBits bits
level F + v - 2 jWord                                               jBits bits
...
level F jWord                             (shortest code)           jBits bits
-------------------------------------------------------------------------------
index tree 1
-------------------------------------------------------------------------------
(same as above)
-------------------------------------------------------------------------------
.
.
.
index tree 2^iBits - 1
-------------------------------------------------------------------------------
(same as above)
-------------------------------------------------------------------------------

index tree 2^iBits (iTree)
-------------------------------------------------------------------------------
# of non-empty levels v in Huffman tree                                  5  bits
first non-empty level F                                                  5  bits
bits used b to represent # leafs L at each level of Huffman tree         5  bits
leafs in level F + v - 1 of Huffman tree  (least freq)                   b  bits
leafs in level F + v - 2 of Huffman tree                                 b  bits
...
leafs in level F of Huffman tree          (most freq)                    b  bits

level F + v - 1 iWord                     (longest code)            iBits bits
level F + v - 2 iWord
iBits bits
...
level F iWord                             (shortest code)           iBits bits
-------------------------------------------------------------------------------
```

```
# bitstrings processed                                    32 bits

encoded iWord                                            variable
encoded jWord                                            variable

encoded iWord                                            variable
encoded jWord                                            variable
...
last encoded iWord                                       variable
last encoded jWord                                       variable
```

LIST OF REFERENCES

1. EI Compendex Web, Academic Science and Engineering Database, http://cpxweb.ei.org

2. Huffman, D., "A Method for the Construction of Minimum Redundancy Codes", Proceedings of the IRE, Vol. 40, pp. 1098-1101, 1952.

3. Fredricksen, H. and Kessler, I., "An Algorithm for Generating Necklaces of Beads in Two Colors", Discrete Mathematics, Vol. 61, pp. 181-188, 1986.

4. Fredricksen, H. M., Maiorana, J. "Necklaces of Beads in k Colors and k-ary de Bruijn Sequences", Discrete Mathematics, Vol. 23, 1978.

5. Shannon, C., "A Mathematical Theory of Communication", Bell Systems Technical Journal, Vol. 27, pp. 379-423, 1948.

6. Nelson, M., "The Data Compression Book", Prentice Hall, 1991.

7. The Data Compression Library, a web compression resource, http://dogma.net/DataCompression/index.shtml

8. Wayner, P., "Compression Algorithms for Real Programmers", Morgan Kaufmann, 2000.

9. Sedgewick, R., "Algorithms", Addison Wesley, 1983.

10. Golomb, S., "Run-Length Encodings" IEEE Transactions on Information Theory Vol. 12, pp. 399-401, July, 1966.

11. Shannon, C., Coding Theorems for a Discrete Source with a Fidelity Criterion, IRE National Convention Records, March 1959, pp. 142-163

12. Rissanen, J., Langdon, G., "Universal Modeling and Coding", IEEE Transactions on Information Theory, Vol. 37, pp. 12-23, January 1981.

13. Rissanen, J., Langdon, G., "Arithmetic Coding", IBM Journal of Research and Development, Vol. 23, pp. 149-162, March 1979.

14. Langdon, G., "An Introduction to Arithmetic Coding", IBM Journal of Research and Development, Vol. 28, pp. 135-149, March 1984.

15. Ziv, J., Lempel, A., "A Universal algorithm for Sequential Data Compression, IEEE Transactions on Information Theory, Vol. 23, pp. 337-343, May 1977.

16. Ziv, J., Lempel, A., "Compression of Individual Sequences Via Variable-rate Coding", IEEE Transactions on Information Theory, Vol 24, pp. 530-536, Sep 1978.

17. Riordan, J, An Introduction to Combinatorial Analysis, Wiley, New York, 1958.

18. Information on Necklaces, Unlabelled Necklaces, Lyndon Words, and DeBruijn Sequences, http://www.theory.csc.uvic.ca/~cos/inf/neck/NecklaceInfo.html

19. The Canterbury Corpus, Benchmark for Lossless Compression Methods, http://corpus.canterbury.ac.nz

20. Waterloo BragZone, Image Compression Resourse, http://links.uwaterloo.ca/bragzone.base.html

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   8725 John J. Kingman Road, Suite 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library 2
   Naval Postgraduate School
   411 Dyer Road
   Monterey, CA 93943-5101

3. Naval Postgraduate School 5
   ATTN: Professor Harold M. Fredricksen
   1411 Cunningham Rd.
   Monterey, CA 93943-5216

4. Captain William L. Crowley Jr 4
   374 Bergin Dr.
   Monterey, CA 93940-4855

5. Department of Mathematics 1
   United States Military Academy
   West Point, New York 10996

6. Copy for NPS Mathematics Department Archives 1
   Prof. Mike A. Morgan
   Department of Mathematics
   Naval Postgraduate School
   411 Dyer Road
   Monterey, CA 93943-5101